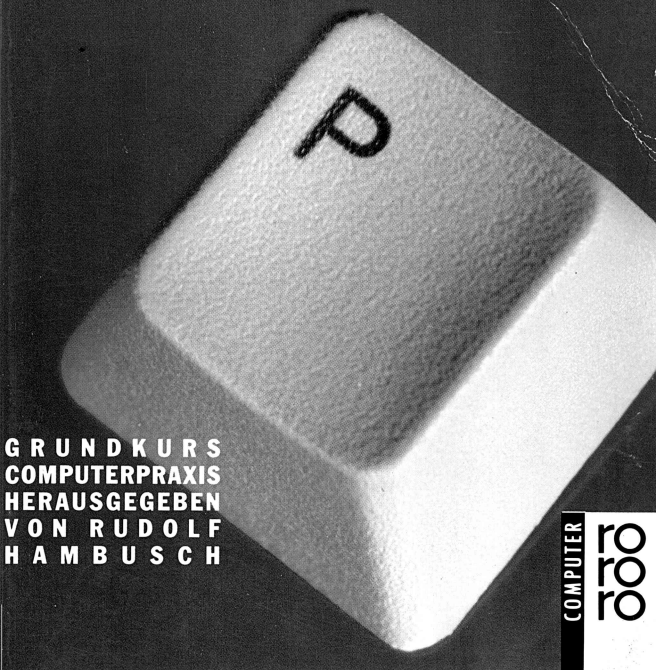
PROGRAMMIERSPRACHE TURBO-PASCAL FÜR FORTGESCHRITTENE

EINE STRUKTURIERTE EINFÜHRUNG

GREGOR KUHLMANN



GRUNDKURS
COMPUTERPRAXIS
HERAUSGEGEBEN
VON RUDOLF
HAMBUSCH

COMPUTER
ro
ro
ro

GREGOR KUHLMANN

PROGRAMMIERSPRACHE TURBO-PASCAL FÜR FORTGESCHRITTENE

EINE STRUKTURIERTE EINFÜHRUNG

GRUNDKURS COMPUTERPRAXIS
HERAUSGEGEBEN
VON RUDOLF HAMBUSCH

ROWOHLT

21.–30. Tausend November 1989

Originalausgabe
Veröffentlicht im Rowohlt Taschenbuch Verlag GmbH,
Reinbek bei Hamburg, Dezember 1988
Copyright © 1988 by Rowohlt Taschenbuch Verlag GmbH,
Reinbek bei Hamburg
Umschlagentwurf Thomas Henning
Satz Times und Helvetica (Linotron 202)
Gesamtherstellung Clausen & Bosse, Leck
Printed in Germany
1680-ISBN 3 499 18155 X

INHALT

EDITORIAL 11

VORWORT 13

1 TURBO-PASCAL 4.0 15

1.1 Bediensystem Turbo-Pascal 4.0 15
1.1.1 Turbo-Pascal starten und beenden 15
1.1.2 Menü-System 16
1.1.3 Eingabe eines Programms (Editieren) 21
1.1.4 Speichern 21
1.1.5 Laden 21
1.1.6 Übersetzen 23
1.1.7 Options 24
1.1.8 Untermenü File 26
1.2 Änderungen und Erweiterungen in Turbo-Pascal 4.0 28
1.2.1 Units 28
1.2.2 Neue Datentypen 30
1.2.3 Veränderungen in den Syntaxvorschriften 30
1.2.4 Veränderungen hinsichtlich Include-Files 31
1.2.5 Veränderungen bei Eingabe-Anweisungen 31
1.2.6 Verkürzte Bedingungsprüfung 32
1.3 Zusammenfassung 32
1.4 Übungen 34
1.5 Lösungen 35

2 DATENSTRUKTUR FELD (ARRAY) 36

2.1 Eindimensionale Datenstruktur 36
2.1.1 Charakteristik des eindimensionalen Feldes 37
2.1.2 Vereinbarung eines Feldes 37

2.1.3 Adressierung eines Feldes 40
2.1.4 Anwendungsbeispiele 42
2.1.5 Zusammenfassung 52
2.1.6 Übungen 52
2.1.7 Lösungen 53
2.2 Zweidimensionale Felder 55
2.2.1 Charakteristik eines zweidimensionalen Feldes 56
2.2.2 Vereinbarung eines zweidimensionalen Feldes 58
2.2.3 Adressierung eines zweidimensionalen Feldes 59
2.2.4 Schleifenkonstruktionen bei Tabellenverarbeitung 62
2.2.5 Zusammenfassung 70
2.2.6 Übungen 71
2.2.7 Lösungen 73

3 DATENSTRUKTUR MENGE (SET) 76

3.1 Vereinbarung von Mengen 77
3.2 Operationen auf Mengen 80
3.2.1 Wertzuweisung 80
3.2.2 Elementprüfung 81
3.2.3 Mengenbildung 81
3.3 Zusammenfassung 84
3.4 Übungen 85
3.5 Lösungen 88

4 DATENSTRUKTUR VERBUND (RECORD) 91

4.1 Charakteristik der Datenstruktur Verbund 91
4.2 Vereinbarung und Zugriff 92
4.3 Zusammengesetzte Datenstrukturen:
ARRAY OF RECORD 95
4.4 Zusammenfassung 103
4.5 Übungen 104
4.6 Lösungen 106

5 DATENSTRUKTUR SEQUENTIELLE DATEI (FILE) 109

5.1 Charakteristik einer sequentiellen Datei 110
5.2 Aufbau und Verarbeitung einer sequentiellen Datei 111

5.2.1 Grundstrukturen eines Dateiverarbeitungs-
programms 111
5.2.2 Vereinbaren einer Datei 114
5.2.3 Öffnen einer Datei 117
5.2.4 Schreiben in eine Datei 118
5.2.5 Lesen aus einer Datei 120
5.2.6 Schließen einer Datei 122
5.2.7 Zusammenfassung 122
5.2.8 Übungen 123
5.2.9 Lösungen 124
5.3 Algorithmen zur Dateibearbeitung 125
5.3.1 Allgemeine Datei-Öffnungsprozedur 126
5.3.2 Suchen in einer Datei 127
5.3.3 Anfügen von Datensätzen 130
5.3.4 Ändern von Datensätzen 136
5.3.5 Sortieren von Datensätzen 142
5.3.6 Löschen von Datensätzen 145
5.3.7 Zusammenstellung des Programms
Lagerverwaltung 150
5.3.8 Zusammenfassung 152
5.3.9 Übungen 153
5.3.10 Lösungen 154

6 INDEX-SEQUENTIELLE DATEI 158

6.1 Kennzeichen einer index-sequentiellen Datei 158
6.1.1 Sequentielle Datei – index-sequentielles
Dateisystem 158
6.1.2 Lesen einer index-sequentiellen Datei 159
6.1.3 Schreiben in eine index-sequentielle Datei 160
6.1.4 Zusammenfassung 160
6.1.5 Übungen 161
6.1.6 Lösungen 161
6.2 Index-sequentielle Dateiorganisation unter
Turbo-Pascal 162
6.2.1 Installationshinweise 162
6.2.2 Steuerungsparameter 164
6.2.3 Zusammenfassung 171
6.2.4 Übungen 172

6.2.5 Lösungen 173
6.3 Einrichten von Satz- und Indexdatei 174
6.3.1 Dateivereinbarung 174
6.3.2 Erzeugen von Satz- und Indexdatei 174
6.3.3 Öffnen von Satz- und Indexdatei 176
6.3.4 Prozeduren zur Anlage und zum Öffnen von Satz- und Indexdatei 176
6.3.5 Schließen der Dateien 178
6.3.6 Zusammenfassung 180
6.3.7 Übungen 181
6.3.8 Lösungen 182
6.4 Eingabe und Ausgabe von Daten bei index-sequentiellen Dateien 183
6.4.1 Arten von Dateiverarbeitungsoperationen 183
6.4.2 Eingabe und Speicherung von neuen Datensätzen 185
6.4.3 Ausgabe einer index-sequentiellen Datei 190
6.4.4 Zusammenfassung 195
6.4.5 Übungen 196
6.4.6 Lösungen 198
6.5 Verändern und Löschen von Datensätzen 203
6.5.1 Verändern von Datensätzen (Update) 203
6.5.2 Löschen von Datensätzen 208
6.5.3 Zusammenstellung der Prozeduren zum Lagerverwaltungsprogramm 210
6.5.4 Zusammenfassung 213
6.5.5 Übungen 214
6.5.6 Lösungen 215
6.6 Behandlung von Dateien mit nicht eindeutigem Schlüssel 218
6.6.1 Anlegen und Öffnen der Indexdatei 218
6.6.2 Schlüsseleintrag in die Indexdatei 219
6.6.3 Suchen in einer Indexdatei mit nicht eindeutigem Schlüssel 221
6.6.4 Zusammenfassung 226
6.6.5 Übungen 226
6.6.6 Lösungen 226

7 DATENSTRUKTUR TEXTDATEI 228

7.1 Charakteristik der Datenstruktur Textdatei 229
7.2 Erstellen einer Textdatei 229
7.3 Lesen einer Textdatei 231
7.4 Einbeziehung von Fehlertexten – Fenstertechnik 233
7.5 Zusammenfassung 238
7.6 Übungen 239
7.7 Lösungen 241

8 DYNAMISCHER DATENTYP ZEIGER (POINTER) 245

8.1 Zeigervariable 246
8.1.1 Vereinbarung und Zugriff auf eine dynamische Variable 246
8.1.2 Speicherplatzfreigabe 248
8.2 Operationen auf Zeigervariablen 249
8.3 Zusammenfassung 252
8.4 Übungen 253
8.5 Lösungen 255

9 DYNAMISCHE DATENSTRUKTUR: VERKETTETE LISTE 256

9.1 Grundstrukturen 257
9.1.1 Charakteristik einer verketteten Liste 257
9.1.2 Vereinbarung der Datenstruktur 259
9.1.3 Algorithmischer Ansatz zum Aufbau einer verketteten Liste 260
9.1.4 Zusammenfassung 267
9.1.5 Übungen 268
9.1.6 Lösungen 270
9.2 Anwendungen 272
9.2.1 Programmrahmen 272
9.2.2 Aufbau einer verketteten Liste durch Anfügen 274
9.2.3 Ausgabe einer verketteten Liste 277
9.2.4 Speichern einer verketteten Liste in eine sequentielle Datei 279

9.2.5 Lesen einer Datei
mit Aufbau einer verketteten Liste 283
9.2.6 Suchen eines Datensatzes
in einer verketteten Liste 284
9.2.7 Sortieren einer verketteten Liste 287
9.2.8 Zusammenstellung der Prozeduren zum
Sportverwaltungsprogramm 299
9.2.9 Zusammenfassung 301
9.2.10 Übungen 303
9.2.11 Lösungen 306

10 ANHANG 314

10.1 ASCII-Tabelle 314
10.2 Literaturverzeichnis 314
10.3 Sachwortregister 316

EDITORIAL

Folgt man den Aussagen von Bildungs- und Wirtschaftspolitikern, so wird das Zusammenleben der Menschen in Zukunft von informationsverarbeitenden Maschinen geprägt sein: Mehr als die Hälfte aller Arbeitenden wird direkt oder indirekt mit Computern zu tun haben. Eine besondere Rolle spielt dabei der heute bereits millionenfach verbreitete persönliche Jedermanncomputer, der Personal Computer (PC). Schüler, Handwerksmeister, Rechtsanwälte, Kaufleute, Lehrer und viele andere werden mit diesem Gerät leben, spielen und arbeiten.
Der Einsatz des persönlichen Computers wird weniger von der Fähigkeit des Benutzers geprägt sein, das Gerät in seiner Technizität (Hardware) zu verstehen, als es mit Hilfe der Computerprogramme (Software) zu bedienen.
Die Serie "Grundkurs Computerpraxis" erklärt Informationsverarbeitung sehr konkret und auf einfache Weise. Dabei steht das, was den Computer im eigentlichen Sinne funktionieren läßt, im Vordergrund: die Software. Sie umfaßt

- Betriebssysteme,
- Anwendungsprogramme,
- Programmiersprachen.

Ausgewählt werden Programme, die sich hunderttausendfach bewährt und einen Standard gesetzt haben, der Gefahr des Veraltens also nur im geringen Maße unterliegen.
Im "Grundkurs Computerpraxis" wird das praktische Computerwissen übersichtlich gegliedert, textverständlich strukturiert, auf das Wesentliche begrenzt und mit Grafiken, Beispielen und Übungen optimal zugänglich gemacht.

Der Herausgeber der Serie, Rudolf Hambusch, leitet am Institut für Schule und Weiterbildung des Landes Nordrhein-Westfalen das Projekt Lehrerfortbildung in Informationsverarbeitung Wirtschaft. Die Autoren sind erfahrene Berufspädagogen, Praktiker und Mitarbeiter im Fortbildungsprojekt.

VORWORT

Die Programmiersprache Turbo-Pascal, schon seit einigen Jahren in der Version 3.x in allen Bereichen der Programmierung sehr weit verbreitet, hat in der Version 4.0 wesentliche Erweiterungen und Verbesserungen erfahren. Damit wurde aber nicht die Version 3.x "außer Kraft" gesetzt. Die beiden Versionen unterscheiden sich in ihrem Bediensystem. In ihrem Befehlsumfang sind sie, bezogen auf die behandelten Themen dieser strukturierten Einführung für Fortgeschrittene, nahezu deckungsgleich.
Die Ausführungen dieses Buches berücksichtigen **beide Versionen** von Turbo-Pascal. Besonderheiten der Version 3.x werden durch **(*3*)**, solche der Version 4.0 durch **(*4*)** kenntlich gemacht.
Der vorliegende Band erscheint in der Reihe "Grundkurs Computerpraxis". Der Herausgeber und die Autoren dieser Reihe haben es sich zum Ziel gesetzt, Interessenten die praktische Arbeit am Computer zugänglich zu machen. Deshalb sind die Darstellungen in diesem Buch problembezogen. Die Inhalte sind auch für "Laien" verständlich und nachvollziehbar.
Die behandelten Pascalbefehle werden stets im Rahmen von vollständigen Programmen ausführlich erläutert. Das ermöglicht es Ihnen, die Wirkungsweise der Befehle direkt am Computer zu testen. Es wird aber auch dargelegt, wie zu den praxisorientierten Problemstellungen die Programme entwickelt werden. Die Kenntnis über Namen und Wirkungsweise von Befehlen reicht nämlich nicht aus. Man muß auch wissen, wie man die Befehle im Sinne einer Problemlösung richtig in ein Programm einbringen muß.
Im Vorgängerband "Programmiersprache Turbo-Pascal" (rororo 8148) werden die Programmstrukturen behandelt. In dem vorliegenden Buch liegt der Schwerpunkt auf **Datenstrukturen**.

Das Buch ist wie folgt aufgebaut:

- Im Kapitel 1 werden das Bediensystem von Turbo-Pascal 4.0 sowie die Veränderungen im Sprachsystem dargestellt, soweit sie für die Thematik des Buches bedeutsam sind.
- Das Kapitel 2 stellt die Datenstruktur Feld (ARRAY) vor. Ein Schwerpunkt liegt in der Behandlung von Tabellen.
- Die Kapitel 3 und 4 befassen sich mit den Datenstrukturen Menge (SET) und Verbund (RECORD).
- Die Kapitel 5, 6 und 7 setzen sich mit dem Aufbau und der Verarbeitung von Dateien (FILE) auseinander: sequentielle Datei, index-sequentielle Datei sowie Textdatei unter Einbeziehung der Fenster-Technik.
- In den Kapiteln 8 und 9 wird die Datenstruktur Zeiger und – darauf aufbauend – die verkettete Liste behandelt.
- Am Ende eines jeden Kapitels finden Sie eine zusammenfassende Übersicht als Merkhilfe. Ein Übungsteil soll Ihnen helfen, das Gelernte zu vertiefen. Zur Kontrolle Ihrer Lösungen werden Ihnen in einem weiteren Abschnitt auch "Musterlösungen" angeboten.

Bei dieser Vorgehensweise können nicht alle Befehle so behandelt werden, wie sie in einem Referenzhandbuch aufgeführt sein müssen. Dazu ist die Programmiersprache Turbo-Pascal zu mächtig. Deshalb ist ein weiterer Folgeband geplant, in dem die Graphik behandelt wird.

Ich danke den Herren Dipl.-Hdl. Gerhard Haase, Dipl.-Hdl. Dipl.-Kfm. Alexander Parkmann, Studienrat Joachim Röhl sowie meinem Sohn Michael für Beratung und Korrektur bei der Erstellung des Manuskriptes.

Gregor Kuhlmann

1 TURBO-PASCAL 4.0

Turbo-Pascal, bereits in der Version 3.x ein Standard in der PC-Welt, ist in der Version 4.0 verbessert und erweitert worden. Hervorzuheben ist die "integrierte Entwicklungsumgebung", das Bediensystem der Version 4.0. Es wird im nächsten Abschnitt vorgestellt. Nicht ganz so "dramatisch", zumindest auf der Ebene einer strukturierten Einführung, sind die Änderungen und Verbesserungen im Sprachsystem. Diese werden im übernächsten Abschnitt behandelt.

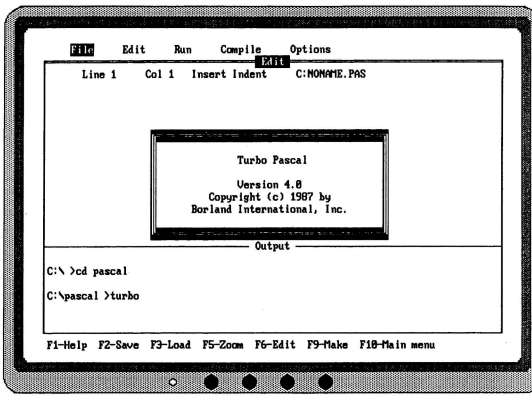
1.1 Bediensystem Turbo-Pascal 4.0

1.1.1 Turbo-Pascal starten und beenden

Starten
Das Programmsystem wird aufgerufen mittels

turbo <RETURN>

Auf dem Bildschirm erscheint daraufhin das Menüsystem von Turbo-Pascal mit einem Fenster, in dem auf das Copyright und die Versionsnummer hingewiesen wird. Wenn Sie irgendeine Taste betätigen, verschwindet das Fenster (siehe Abbildung Seite 16).



Beenden
Für den "Ausstieg" aus dem Turbo-Pascal-System gibt es verschiedene Kommandos. Der schnellste Weg geht über die Tastenkombination <ALT> <X>.

1.1.2 Menü-System

Bildschirmaufbau
Der Bildschirm ist in vier Abschnitte unterteilt. Am oberen Bildschirmrand ist die Menüleiste, darunter die Statuszeile zu sehen. Im mittleren Teil des Bildschirms befinden sich das Edit- und das Output-Fenster. Am unteren Rand des Bildschirms schließlich wird in der sogenannten Hotkey-Leiste angezeigt, welche Funktionen über die Funktionstasten unmittelbar ausgeführt werden können.
Probieren Sie es aus! Drücken Sie die Funktionstaste <F5>!
Wirkung: Das Output-Fenster verschwindet. Das Edit-Fenster erstreckt sich nun über den ganzen Bildschirm.

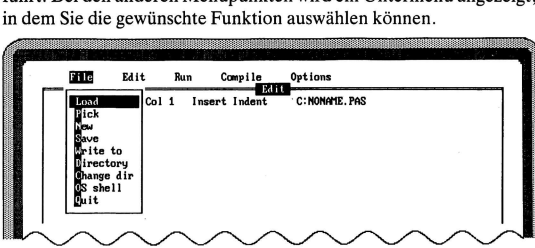
Menüleiste

Nach dem Start des Turbo-Pascal-Systems und dem Ausblenden des Copyright–Fensters sehen Sie in der Menüleiste, daß das Wort **File** hervorgehoben ist. Das System ist somit bereit, Kommandos auszuführen, die Sie im Hauptmenü anwählen.

Die einzelnen Menüpunkte können Sie auf unterschiedliche Weise ansprechen:

- Geben Sie den hervorgehobenen Anfangsbuchstaben des Menüpunktes ein oder
- bewegen Sie mit Hilfe der Pfeiltasten <→> oder <←> den hervorhebenden Balken auf den gewünschten Menüpunkt und betätigen Sie die RETURN-Taste!

Bei den Menüpunkten **Edit** und **Run** wird die Funktion sofort ausgeführt. Bei den anderen Menüpunkten wird ein Untermenü angezeigt, in dem Sie die gewünschte Funktion auswählen können.



Zur Anwahl einer Funktion innerhalb eines Untermenüs haben Sie wiederum zwei Möglichkeiten. Sie rufen die gewünschte Funktion auf, indem Sie entweder

- den hervorgehobenen Anfangsbuchstaben des Untermenüpunktes eingeben oder
- mit Hilfe der Pfeiltasten <↑> bzw. <↓> den hervorhebenden Balken auf die gewünschte Funktion bewegen und die RETURN-Taste betätigen.

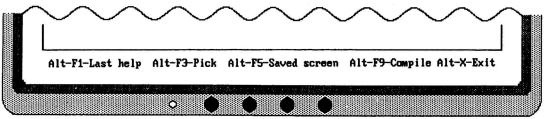
Beispiel:

Probieren Sie den Aufruf des Untermenüs zum Auswahlpunkt File! Geben Sie <F> ein! Am Bildschirm erscheint obenstehende Darstellung.

Mittels <ESC>-Taste bzw. <EingLösch>-Taste oder <F10> kehren Sie aus dem Untermenü in das Hauptmenü zurück.

Funktionstastenleiste

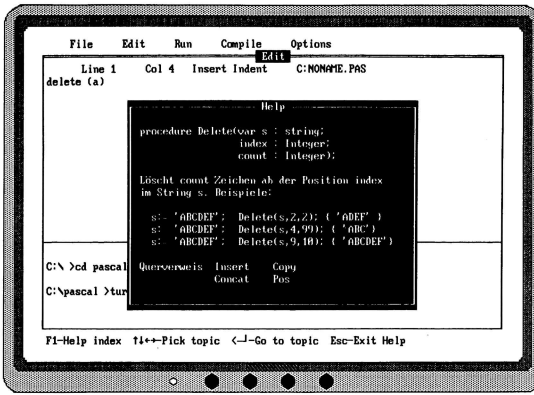
Über die Funktionstasten, die am unteren Bildschirmrand angezeigt werden, können Sie häufig benötigte Funktionen unmittelbar aufrufen. Wenn Sie die <ALT>-Taste einige Sekunden festhalten, sehen Sie, daß die Funktionstasten in der Kombination mit der <ALT>-Taste doppelt belegt sind.



Mit der Funktionstaste <F1> können Sie erläuternde Hilfetexte zu den Menüpunkten aufrufen. Mit <ESC> wird der Hilfetext wieder weggeblendet.

Wenn Sie sich im Editor befinden, können Sie sich durch die Tastenkombination <CTRL> <F1> Hilfetexte zu Standard-Prozeduren und -Funktionen sowie zu den Units auf den Bildschirm holen. Units sind Programmbibliotheken innerhalb des Turbo-Pascal-Systems. Wie die Hilfefunktion bei der Eingabe oder Bearbeitung eines Quellprogramms aufgerufen wird, sei an einem kleinen Beispiel erläutert:

Sollten Sie sich noch nicht im Editor befinden, so aktivieren Sie ihn mit <ALT> <E> oder <F6>. Nun geben Sie beispielsweise den Namen der Stringprozedur **delete** ein. Fahren Sie den Cursor unter einen beliebigen Buchstaben des Wortes delete und drücken Sie die Tastenkombination <CTRL> <F1>. Am Bildschirm erscheint in einem Fenster folgender Hilfetext:



Innerhalb des Textfensters können Sie auch die dort angegebenen Querverweise anwählen. Dazu bewegen Sie mit Hilfe der Pfeiltasten den Balken auf die gewünschte Stelle und betätigen die Taste <RETURN>. Der gewünschte Hilfetext wird nun im Fenster angezeigt.
Möchten Sie in das vorhergehende Fenster zurückkehren, drücken Sie die Tastenkombination <ALT> <F1>.
Mit <ESC> bzw. <EingLösch> verlassen Sie die Hilfefunktion und kehren in den Editor zurück.
Die weiteren Funktionstasten sind mit folgenden Funktionen belegt:

Taste	Bezeichnung	Funktion
<F2>	Save	speichert das im Editor befindliche Quellprogramm auf externen Speicher
<F3>	Load	lädt ein Quellprogramm vom externen Speicher in den Editor
<F5>	Zoom	vergrößert oder verkleinert das Edit-Fenster oder das Output-Fenster
<F6>	Edit	schaltet vom Output-Fenster auf das Edit-Fenster um und umgekehrt
<F9>	Make	startet die Compilierung des Programms, das sich im Editor befindet
<F10>	Main menu	verläßt den Editor und schaltet um auf die Menüleiste

Tastenkombinationen mit der <ALT>-Taste (Hot Keys)
In Kombination mit der <ALT>-Taste und einer weiteren Taste können Sie die Menüpunkte der Menüleiste unmittelbar anwählen.

Tastenkombination	Funktion
<ALT> <C>	ruft den Compiler auf
<ALT> <E>	aktiviert den Editor
<ALT> <F>	ruft das Menü File auf
<ALT> <O>	aktiviert das Menü Options
<ALT> <R>	bewirkt den Start eines Programms
<ALT> <X>	beendet Turbo-Pascal. Das System kehrt auf die Betriebssystemebene zurück.

1.1.3 Eingabe eines Programms (Editieren)

Um ein Programm eingeben zu können, muß der Editor gestartet werden. Auch dazu gibt es verschiedene Möglichkeiten. Am schnellsten geht es mit Hilfe der Funktionstaste <F6>.
Nun können Sie das Quellprogramm eingeben. Die einzelnen Editierkommandos werden hier nicht dargestellt, weil sie mit den Kommandos der Version 3.x übereinstimmen. Diese sind im ersten Band beschrieben worden.
Wollen Sie das Editieren beenden, so stehen Ihnen dazu wiederum mehrere Kommandos zur Verfügung:

- Mittels der Funktionstaste <F10> verlassen Sie den Editor und schalten um auf die Menüleiste. In ihr können Sie nun die gewünschte Funktion anwählen.
- Es geht aber noch direkter: Wenn Sie das eingegebene Programm unmittelbar übersetzen und ausführen lassen möchten, so beenden Sie das Editieren mit der Tastenkombination <ALT> <R>.
- Möchten Sie das Programm zunächst nur übersetzen lassen, rufen Sie mittels <ALT> <C> den Menüpunkt Compiler auf und starten mit <RETURN> den Übersetzungslauf.

1.1.4 Speichern

Mit der Funktionstaste <F2> veranlassen Sie die Speicherung des Programms. Wenn Sie noch keinen Dateinamen für Ihr Programm vergeben haben, so fordert das System Sie in einem kleinen Fenster zur Eingabe des Dateinamens auf. Die Regeln für die Schreibweise des Dateinamens entsprechen denen der Version 3.x.

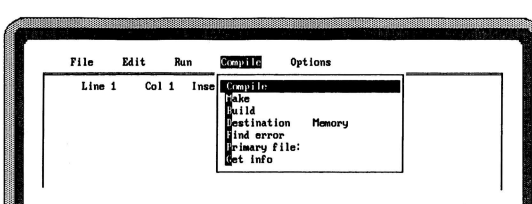
1.1.5 Laden

Um ein Quellprogramm in das Editorfenster zu laden, betätigen Sie die Funktionstaste <F3>. In einem Fenster wird Ihnen der Name der Datei, mit der Sie gerade vorher gearbeitet haben, vorgeschlagen. Ist es die gewünschte Datei, so bestätigen Sie mit <RETURN> den Vorschlag.
Möchten Sie mit einer anderen Datei arbeiten, so geben Sie ihren Namen ein. Der vorgeschlagene Dateiname verschwindet nach der Eingabe des ersten Buchstabens.
Falls Sie den Dateinamen nicht mehr wissen, tippen Sie auf die Leer-

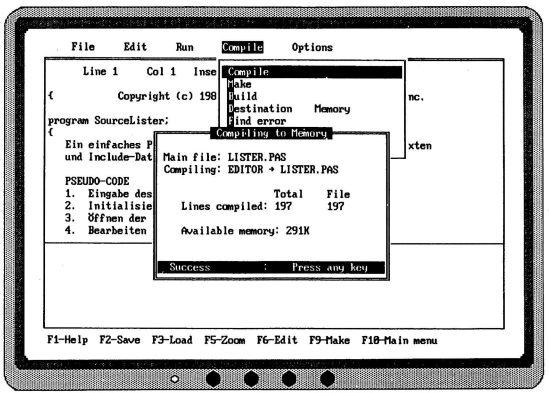
taste. Der vorgeschlagene Dateiname verschwindet. Nach Betätigung der RETURN-Taste werden in einem weiteren Fenster alle Dateinamen mit der Kennung .PAS sowie die Unterverzeichnisse, die im aktuellen Directory verzeichnet sind, angezeigt. Mit Hilfe der Pfeiltasten stellen Sie den Balken auf den Namen der gewünschten Datei und drücken die RETURN-Taste.
Sie können für die Anzeige der Dateinamen eine Vorauswahl treffen. Kommen beispielsweise nur Dateinamen in Frage, die mit "Zei" beginnen, so geben Sie **Zei*** ein. Es werden Ihnen dann nur die Dateinamen zur Auswahl gestellt, deren ersten drei Buchstaben aus "Zei" bestehen und deren Dateikennung "pas" ist.
Falls Sie gerade vorher an einem Programmtext gearbeitet haben, der noch nicht gespeichert ist, fragt das System Sie, ob der im Editor-Fenster befindliche Text gespeichert werden soll. Mit y(es) veranlassen Sie die Speicherung. Mit n(o) wird der Text nicht gespeichert. Anschließend wird der Inhalt der gewünschten Datei geladen und im Editor–Fenster angezeigt.

1.1.6 Übersetzen

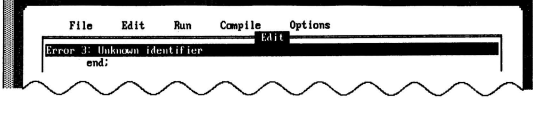
Wie bereits oben kurz erwähnt, kann der Übersetzungslauf aus dem Editor durch <F9>, <ALT> <R> oder auch durch <ALT> <C> aufgerufen werden. Dabei muß sich das zu übersetzende Programm im Editor-Fenster befinden.
Wenn Sie <ALT> <C> eingegeben haben, wird Ihnen folgendes Menü angezeigt:



Von den im Untermenü enthaltenen Funktionen sind im Rahmen einer Einführung aber nur die Funktionen **Compile** und **Destination** von Bedeutung.
Unter dem Auswahlpunkt Destination können Sie bestimmen, ob das übersetzte Programm (Compilat) nur im Arbeitsspeicher aufgebaut oder ob es als sogenannter .EXE-File auf der Diskette oder Platte abgelegt werden soll. Mit .EXE-File wird ein Programm bezeichnet, das unmittelbar aus der DOS-Ebene aufgerufen werden kann. Für die Ausführung des Programms wird dann nicht mehr die Pascal-Sprachumgebung benötigt.
Die Voreinstellung steht auf **Memory**. Das Compilat wird somit nur im Arbeitsspeicher aufgebaut. Möchten Sie das Compilat auf der Diskette oder Magnetplatte speichern, wählen Sie Destination an. Durch Betätigen der RETURN-Taste schalten Sie auf Disk um. Nun wird bei jeder Compilation eine .EXE-Datei auf dem externen Datenträger erzeugt. Lautete die Datei Ihres Quellprogramms beispielsweise Zeiger .PAS, so wird das übersetzte Programm unter Zeiger.EXE auf dem externen Datenträger abgelegt.
Die Compilation wird aus dem Untermenü durch Eingabe von <C> ausgelöst. Ist das Programm syntaktisch fehlerfrei, erscheint die Meldung:



Bei <ALT> <R> wird das im Editor-Fenster befindliche Programm übersetzt und unmittelbar ausgeführt, sofern es syntaktisch fehlerfrei ist. Das compilierte Programm befindet sich im Arbeitsspeicher, sofern im Untermenü zu Compile unter Destination nichts anderes eingetragen war.
Stellt der Compiler einen syntaktischen Fehler fest, kehrt das System in den Editor zurück. Die fehlerhafte Stelle im Programm wird angezeigt. Am oberen Bildschirmrand wird der Fehler erläutert.



Nach Betätigen einer Taste verschwindet die Fehlermeldung. Sie können sie sich durch die Tastenkombination <CTRL> <Q> <W> wieder anzeigen lassen.

1.1.7 Options

Von den Auswahlpunkten des Options-Menüs, das aus dem Editor mittels <ALT> <O> aufgerufen wird, ist im Rahmen einer Einführung vor allem der Auswahlpunkt Compiler von Bedeutung. Im Untermenü Compiler interessieren folgende Punkte:

Range checking
Steht Range checking auf off, so wird nicht geprüft, ob bei
- Zugriff auf einen String bzw. Array,
- Zuweisungen auf Variablen von Aufzählungs- oder Unterbereichstypen oder
- Prozeduraufrufen mit Werteparametern

die vorher vereinbarten Grenzen eingehalten worden sind.
Ist Range checking auf on gesetzt, wird das Programm bei einer Bereichsüberschreitung mit einem Fehlerkommentar abgebrochen.
In der Version 3.x kann der Range-checking-Schalter nur durch die Compileranweisung (*$R+*) auf on gesetzt werden.

Stack checking
Stack checking wird in der Regel nur bei großen Programmen benötigt. Steht Stack checking auf on, prüft das System bei der Ausführung eines Programms vor jedem Prozedur- und Funktionsaufruf, ob noch genügend Platz zur Speicherung der erforderlichen Systeminformationen vorhanden ist.
In der Version 3.x kann die Stack-Prüfung nur durch (*$K+*) eingeschaltet werden.

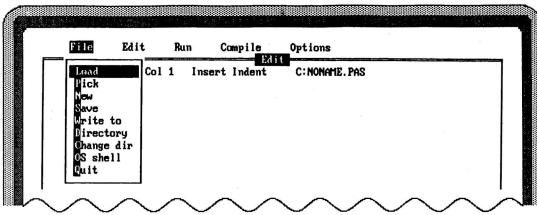
I/O checking
Steht I/O checking auf on, wird die Ausführung eines Programms beendet, wenn eine Ein-/Ausgabeoperation nicht korrekt ausgeführt worden ist. Wird beispielsweise eine Dezimalzahl in eine INTEGER-Variable eingegeben, bricht das Programm mit einer Fehlermeldung ab.
Soll ein solcher Programmabbruch unterbunden werden, ist I/O checking auf off zu setzen. Dann muß der Programmierer durch Einbeziehung der Funktion IORESULT in das Programm für die erforderliche Fehlerbehandlung sorgen.
In der Version 3.0 kann die Fehlerprüfung nur durch die Compileranweisung (*$I–*) abgeschaltet werden.

Das I/O checking muß zumindest in professionellen Programmen bei jeder Eingabe, die zu einem Programmabbruch führen könnte, angewendet werden.

Var-string checking
Bei der Übergabe von Variablen an Prozeduren muß normalerweise der Typ des aktuellen Parameters genau mit dem Typ des formalen Parameters übereinstimmen. Die strikte Typüberprüfung kann bei String-Variablen gelockert werden, indem Var-string checking auf **relaxed** gesetzt wird. Es kann dann beispielsweise eine String-Variable vom Typ string (.30.) an einen formalen Parameter übergeben werden, der mit einer Länge von 40 Stellen festgelegt ist.
In der Version 3.x kann die strikte Typüberprüfung nur durch die Compileranweisung (*$V–*) abgeschaltet werden.

1.1.8 Untermenü File

Weitere wichtige Funktionen sind nur über das Untermenü zu dem Auswahlpunkt File zu erreichen.



New

New löscht den Inhalt des Edit-Fensters und vergibt als vorläufigen Dateinamen den Namen NONAME.PAS. Das Editorprogramm wird aufgerufen, und ein Programm kann neu eingegeben werden.
Falls sich noch ein Text im Editor befindet, der seit der letzten Veränderung noch nicht gespeichert wurde, werden Sie gefragt, ob Sie den Text speichern wollen. Geben Sie y(es) ein, so wird der Text gespeichert. Bei n(o) unterbleibt die Speicherung.
Für den jetzt einzugebenden Programmtext wird der "endgültige" Dateiname erst vereinbart, wenn Sie den Text speichern.

Write to

Mit der Funktion Write to können Sie den Programmtext, der sich gerade im Editor befindet, in eine Datei speichern. Wenn Sie die Funktion aufrufen, werden Sie nach dem Namen der Datei gefragt, unter dem die Speicherung erfolgen soll.

Beispiel:
Sie haben ein Programm Test1.pas geladen und einige Änderungen in dem Programm vorgenommen. Wenn Sie das geänderte Programm nicht wieder unter Test1.pas, sondern unter einem anderen Namen speichern möchten, so verwenden Sie die Funktion Write to.

Directory
Über diese Funktion lassen Sie sich das Inhaltsverzeichnis des aktuellen Directorys anzeigen.
Durch Angabe einer Suchmaske, z. B. Tes*.*, erfolgt eine Vorauswahl der Dateinamen, die angezeigt werden sollen.
Wollen Sie den Dateinamen nicht in dem aktuellen, sondern in einem anderen Verzeichnis suchen, so stellen Sie den Balken mit Hilfe der Pfeiltasten auf das gewünschte Directory. Durch Betätigen der RETURN-Taste wird der Inhalt des angewählten Directorys angezeigt. Auch von diesem angezeigten Directory können Sie in die Anzeige eines weiteren Verzeichnisses umschalten, indem Sie den Lichtbalken auf den gewünschten Verzeichnisnamen stellen und die RETURN-Taste betätigen. Auf diese Weise ist es Ihnen möglich, die verschiedensten Inhaltsverzeichnisse "durchzublättern". Nach Drücken der <ESC>-Taste oder <EingLösch>-Taste verschwinden die aufgebauten Verzeichnis-Fenster.
Sie können aber auch den Namen des Verzeichnisses, in dem Sie nachschauen wollen, unmittelbar eingeben. Dazu müssen Sie die Funktionstaste <F4> betätigen. In das auf dem Bildschirm angezeigte Eingabefenster ist dann der vollständige Suchpfad einzugeben.

Change dir
Mit der Funktion Change dir können Sie in ein anderes Directory wechseln, das dann als Standard-Directory angesehen wird.

OS shell
Mit der Funktion OS shell schalten Sie um auf die Betriebssystemebene. Sie können dann, ohne Turbo-Pascal zu beenden, DOS-Kommandos ausführen lassen.

Beispiel:
Vor dem Speichern eines Programms wollen Sie auf der Diskette Platz schaffen. Dazu schalten Sie durch <ALT> <F> <O> auf die DOS-Ebene um und löschen die nicht mehr benötigten Dateien. Anschließend kehren Sie durch Eingabe von **exit** <RETURN> auf die Turbo-Pascal-Ebene zurück.

Quit
Die Funktion Quit hat dieselbe Wirkung wie <ALT> <X>. Sie verlassen damit Turbo-Pascal und gelangen auf die MS-DOS-Ebene.

Pick
Die Funktion Pick ist dann sehr hilfreich, wenn Sie mit mehreren Quellprogramm-Dateien abwechselnd arbeiten. Das System merkt sich, mit welchen Dateien Sie gearbeitet haben. Wenn Sie von der Bearbeitung eines Quellprogramms in ein anderes wechseln wollen, werden Ihnen beim Aufruf dieser Funktion die Namen der zuletzt bearbeiteten Dateien angezeigt. Mit Hilfe der Pfeiltasten können Sie den Balken auf die nunmehr gewünschte Datei stellen. Durch Betätigen der RETURN-Taste wird die Datei geladen.
Die Pick-Funktion wird auch durch <ALT> <F3> direkt aus dem Editor angesprochen.

1.2 Änderungen und Erweiterungen in Turbo-Pascal 4.0

1.2.1 Units

Das Turbo-Pascal System 4.0 enthält 7 Standard-Units. Eine Unit ist eine Prozedurbibliothek, in der die vordefinierten Datentypen, Konstanten, Variablen, Prozeduren und Funktionen für ein bestimmtes Aufgabengebiet zusammengefaßt sind. So sind beispielsweise alle Systemroutinen, die für die Ausgabe von Daten über einen Drucker benötigt werden, in der Unit **Printer** zusammengestellt. Wollen Sie in Ihrem Programm Daten ausdrucken lassen, so müssen Sie den Compiler durch **uses Printer;** anweisen, beim Compilieren die dafür erforderlichen Systemroutinen in Ihr Programm einzubinden. Haben Sie diese uses-Anweisung vergessen, bringt der Compiler bei dem Versuch, die Anweisung

```
writeln (lst, 'Ausgabe irgendwas');
```

zu übersetzen, die Fehlermeldung, daß ihm der Name **lst** nicht bekannt ist.
In der folgenden Tabelle sind die Units und ihre Funktionen aufgelistet.

Unit	Funktion
System	Beinhaltet die Grundfunktionen von Turbo-Pascal. Es wird beim Compilieren automatisch eingeladen. Ein **Aufruf** durch eine uses-Anweisung ist **nicht erlaubt.**
Crt	Behandelt die Routinen für Bildsschirm und Tastatur.
Printer	Enthält die Routinen für Datenausgabe über Drukker.
Dos	Stellt die Schnittstelle zum Betriebssystem dar. Es wird benötigt, wenn MS-DOS-Funktionen aus einem Pascal-Programm aufgerufen werden sollen.
Graph	Enthält ein komplettes Graphik-Paket.
Turbo3	Beinhaltet einen Teil von Systemroutinen der Version 3.x, die in der Version 4.0 nicht mehr enthalten sind, wie z. B. kbd.
Graph 3	Enthält sämtliche Graphikfunktionen der Version 3.x.

1.2.2 Neue Datentypen

Gegenüber der Version 3.x, die nur die beiden INTEGER-Typen byte und integer kannte, werden in der Version 4.0 noch drei weitere INTEGER-Typen zur Verfügung gestellt.

Typ	Bereich	Format
ShortInt	− 128 ...+ 127	1 Byte mit Vorzeichen
LongInt	−2147483648 ... +2147483647	4 Bytes mit Vorzeichen
Word	0 ... 65353	2 Bytes vorzeichenlos

Wenn Ihr Rechner mit einem mathematischen Co-Prozessor ausgerüstet ist, können Sie die Erweiterung der REAL-Typen um Single, Double, Extended und Comp nutzen, die eine höhere Rechengenauigkeit ermöglichen. Nähere Einzelheiten entnehmen Sie bitte dem Benutzerhandbuch.
Die Handhabung des Datentyps **STRING** ist vereinfacht worden. Wird hinter einer Variablen nur **string** als Datentyp angegeben, so wird die Variable mit einer Länge von 255 Zeichen angelegt. Die Vereinbarung **string** entspricht der Vereinbarung **string (.255.)**. Der Datentyp string darf nun auch in der Deklarierung formaler Prozedurparameter verwendet werden.

1.2.3 Veränderungen in den Syntaxvorschriften

Die Version 4.0 ist in einigen Punkten pedantischer als die Version 3.x. Sie hält sich stärker an den ANSI-Standard von Pascal. Im folgenden werden die im Rahmen einer Einführung wesentlichen Veränderung kurz skizziert.

- Der Name, der im Programmkopf verwendet wird, darf im Programm nicht noch einmal benutzt werden.
- Die Kompatibilität einer CHAR-Variablen mit einer Variablen vom Typ string (.1.) ist aufgehoben worden.

Beispiel:

```
program Test;
var A : char;
    B : string (.1.);
begin
  A := 'x';
  B := A;          (* zulässige Zuweisung *)
  A := B;          (* unzulässig! Keine Typkompatibilität *)
end.
```

- Eine Laufvariable in einer FOR-Anweisung muß entweder global oder lokal vereinbart sein. Sie darf also nicht gleichzeitig sowohl in der globalen als auch in der lokalen Variablenvereinbarung aufgeführt sein.

1.2.4 Veränderungen hinsichtlich Include-Files

- Der Aufruf eines Include-Files (*$I Dateiname *) darf nicht innerhalb des Anweisungsteils, also nicht zwischen **begin** und **end**, erfolgen.
- Zur Schreibweise: $I und Dateiname müssen durch ein Leerzeichen getrennt sein.
- Beinhaltet der Include-File ausführbare Pascal-Befehle, so müssen sie eine komplette Prozedur bzw. Funktion darstellen.
- Aus einem Include-File können weitere Include-Files aufgerufen werden. Die Verschachtelungstiefe beträgt maximal 8 Ebenen.

1.2.5 Veränderungen bei Eingabe-Anweisungen

- Bei einer Anweisung in der Form readln (Wert), wobei Wert als INTEGER-Variable vereinbart ist, erwartet Turbo-Pascal unbedingt die Eingabe einer Zahl. In der Version 3.x ist es möglich, den Programmlauf durch Betätigen der RETURN-Taste fortzusetzen, ohne den Inhalt der Variablen Wert zu verändern.

- Das direkte Lesen eines Zeichens von der Tastatur ohne Echo auf dem Bildschirm wurde in der Version 3.x durch

  read (kbd, Zeichen) (var Zeichen: char)

  bewirkt. In der Version 4.0 wird diese Anweisung durch

  Zeichen :=ReadKey

  ersetzt.

1.2.6 Verkürzte Bedingungsprüfung

In einem zusammengesetzten Bedingungsausdruck wird die Bedingungsprüfung abgebrochen, wenn das Ergebnis feststeht.

Beispiel:

```
if (A = 0) or (B/A = 1)
then...
```

Wenn die Variable A den Wert Null aufweist, wird der zweite Teil der Bedingung nicht mehr behandelt. Diese Vorgehensweise wird Kurzschlußverfahren genannt. Im Gegensatz dazu wird in der Version 3.x stets der komplette Bedingungsausdruck geprüft. Das hätte in dem obigen Beispiel wegen einer Division durch null einen Programmabbruch zur Folge.
Wird aber eine vollständige Bedingungsprüfung gewünscht, so ist das durch die Compiler-Anweisung (*$B+*) zu erreichen. Die Voreinstellung des Compilers steht auf (*$B−*), auf Kurzschlußverfahren.

1.3 Zusammenfassung

- Die Kommandos des Turbo-Pascal-Systems können auf verschiedene Weise aufgerufen werden:

Beschreibung der Funktion		Kommando
Start des Turbo-Pascal-Systems		turbo <RETURN>
Beenden des Turbo-Pascal-Systems		<ALT> <X> <ALT> <F> <Q> <F10> <F> <Q>
Aufruf des Editors		<F10> <E> <F6>
	mit Löschen Bildschirm aus einem Untermenü	<F10> <F> <N> <ALT> <E>
Editieren beenden	mit Umschaltung ins Hauptmenü	<F10>
	mit Aufruf des Compilers	<F10> <C> <C> <ALT> <C> <C> <F9>
	mit Start des Programms	<F10> <R> <ALT> <R>
	mit Speichern des Programms	<F10> <F> <S> <ALT> <F> <S>
Quellprogramm (zwischen-)speichern		<F2>
Quellprogramm laden		<F10> <F> <L> <ALT> <F> <L> <F3>
Anzeige des Inhaltsverzeichnisses		<F10> <F> <D> <ALT> <F> <D>
Directory ändern		<F10> <F> <C> <ALT> <F> <C>
Aufruf der Betriebssystemebene Rückkehr zu Turbo-Pascal durch		<F10> <F> <O> exit <RETURN>

- Im Menü Options können Compileranweisungen eingestellt werden.
- Das Compilat wird auf der Diskette bzw. Magnetplatte abgelegt, wenn im Menü Compile unter Destination Disk eingestellt ist.
- Mit der <ESC>-Taste bzw. <EingLösch>-Taste wird ein Menü wieder verlassen.

- Beim Editieren eines Programmtextes kann man durch <CTRL> <F1> zu Standard-Prozeduren und -Funktionen Hilfetexte anfordern. Mittels <ESC> bzw. <EingLösch> kehrt man in den Editor zurück.
- Bedingungsprüfungen werden gemäß Voreinstellung des Compilers im Kurzschlußverfahren durchgeführt.
- Turbo-Pascal verfügt in der Version 4.0 über 5 INTEGER-Typen: Byte, ShortInt, Longint, Word und Integer.

1.4 Übungen

1. Aufgabe
Wie wird der Editor am einfachsten aktiviert?

2. Aufgabe
Sie wollen in einem Programm die Stringfunktion **copy** benutzen, haben jedoch vergessen, wie die Parameter anzugeben sind. Wie können Sie sich am schnellsten Informationen darüber besorgen?

3. Aufgabe
Sie haben ein Programm eingegeben und wollen es nun übersetzen lassen. Dabei soll das Compilat als .EXE-File auf der Diskette bzw. Magnetplatte gespeichert werden. Welche Bedienungsschritte müssen Sie dazu vornehmen?

4. Aufgabe
In welchem Menüpunkt der Menüleiste werden Compileranweisungen eingestellt?

5. Aufgabe
Wodurch unterscheidet sich der Datentyp Byte vom Datentyp ShortInt?

1.5 Lösungen

1. Aufgabe
Der Editor wird am einfachsten über <ALT> <E> oder <F6> aktiviert.

2. Aufgabe
Geben Sie das Wort **copy** ein und stellen Sie den Cursor auf den Anfang des Wortes! Anschließend betätigen Sie die Tastenkombination <CTRL> <F1>. Durch <ESC> bzw. <EingLösch> wird der Hilfetext wieder weggeblendet.

3. Aufgabe
Um das Compilat auch als .EXE-File abzulegen, rufen Sie mittels <ALT> <C> das Menü Compile auf. Durch Eingabe von <D> wählen Sie den Menüpunkt Destination an. Wird hinter Destination Memory angezeigt, so drücken Sie die <RETURN>-Taste. Dadurch wird der Eintrag hinter Destination auf Disk umgeschaltet. Mit <ESC> bzw. <EingLösch> blenden Sie das Fenster wieder weg. Anschließend betätigen Sie die Taste <C>. Damit starten Sie den Übersetzungslauf.

4. Aufgabe
Compileranweisungen werden im Menüpunkt Options eingestellt.

5. Aufgabe
Der Wertebereich von Byte reicht von 0 bis 255. ShortInt hat einen Wertebereich von −128 bis +127.

2 DATENSTRUKTUR FELD (ARRAY)

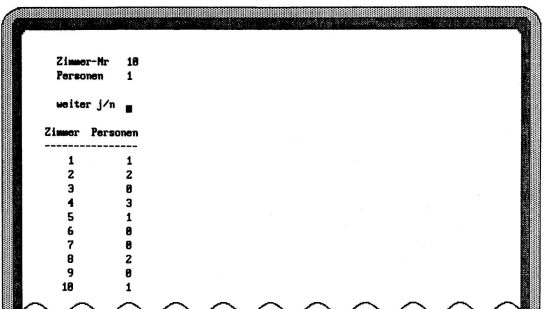
Bei vielen Problemstellungen wäre eine Lösung nur sehr schwer und umständlich zu programmieren, wenn nur die einfachen Datentypen INTEGER, REAL, CHAR oder BOOLEAN zur Verfügung stünden. Deshalb bietet PASCAL wie auch andere höhere Programmiersprachen die Möglichkeit, aus den einfachen Datentypen weitere Datentypen aufzubauen. Ein Datentyp, der sich aus anderen Datentypen zusammensetzt, wird **strukturierter Datentyp** oder **Datenstruktur** genannt. In Abhängigkeit davon, wie die Datenstruktur gebildet wird, unterscheidet man zwischen

Feld : array
Menge : set
Satz : record
Datei : file

Zunächst wird der strukturierte Datentyp Feld behandelt.

2.1 Eindimensionale Datenstruktur

Einführungsbeispiel: Ein Hotel möchte mittels Computer die Zimmerverwaltung vereinfachen. Auf dem Bildschirm wird die Zimmerbelegung angezeigt. Wenn ein Zimmer belegt wird, gibt der Mitarbeiter die Zimmernummer sowie die Anzahl der Personen, die in dem Zimmer schlafen werden, über die Tastatur ein. Anschließend zeigt der Rechner die aktualisierte Zimmerbelegung auf dem Bildschirm an.

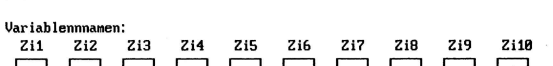


Wenn Sie sofort sehen wollen, wie das Programm abläuft, so geben Sie es ein. Es ist auf Seite 44 abgebildet. Der Ablauf des Programms ist relativ klar:

Grob-Algorithmus Zimmerbelegung
Eingabe Zimmernummer
Eingabe Personenzahl
Eintragen der Personenzahl unter der entsprechenden Zimmernummer
bis Ende

2.1.1 Charakteristik des eindimensionalen Feldes

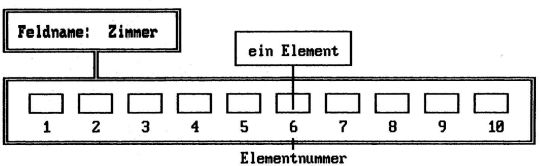
Wie aber wird die Zimmerbelegung im Arbeitsspeicher des Rechners dargestellt? Man könnte für jedes einzelne Zimmer eine Integer-Variable vereinbaren, in die die Anzahl der Personen eingetragen wird.



Das würde jedoch zu einer sehr umständlichen Programmierung führen. Man stelle sich den Programmieraufwand vor, um dem Rechner mitteilen zu können, mit wieviel Personen ein bestimmtes Zimmer belegt wird:

```
readln (Nr);
readln (Anzahl);
case Nr of
1: Zi1 := Anzahl;
2: Zi2 := Anzahl;
3: Zi3 := Anzahl;
...
10: Zi10 := Anzahl;
end;
```

Eleganter und einfacher wird die Programmierung, wenn die Zimmerbelegung in einem Feld dargestellt wird.



In einem Feld können mehrere Elemente zu einer neuen Einheit zusammengefaßt werden. Voraussetzung ist jedoch, daß die Elemente den **gleichen Datentyp** haben.

> Ein Feld besteht aus der Zusammenfassung mehrerer Elemente gleichen Datentyps.

Diese Datenstruktur wird als **eindimensional** charakterisiert, weil sich ihr Aufbau in einer Dimension, d. h. in einer Richtung vollzieht.

2.1.2 Vereinbarung eines Feldes

In unserem Beispiel sollen die 10 INTEGER-Variablen Zi1 bis Zi10 zu einem Feld unter dem Namen Zimmer zusammengefaßt werden. Dazu ist im Vereinbarungsteil anzugeben:

var Zimmer: **array** (. 1..10 .) of integer;

Das Schlüsselwort zur Vereinbarung von Feldern lautet **array** (engl.: Reihe, Feld). Dahinter folgt in runden Klammern mit Punkt oder eckigen Klammern die Angabe der ersten und der letzten Elementnummer. Zwischen den beiden Elementnummern stehen zwei (!) Punkte. Hinter **of** wird vermerkt, welchen Datentyp das Feld hat. Allgemein lautet die Vereinbarung von Feldern:

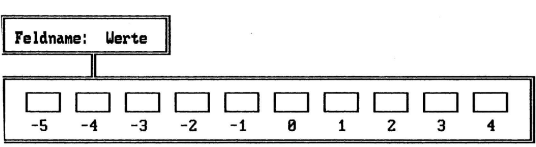
var Feldname: array (.Untergrenze .. Obergrenze .) of Datentyp

Die Angaben zu Untergrenze und Obergrenze müssen **Konstanten ordinalen Typs** sein. REAL-Werte sind also nicht zulässig. Der Datentyp des Feldes ist dagegen beliebig.
Der Wert der Untergrenze muß kleiner sein als der Wert der Obergrenze.

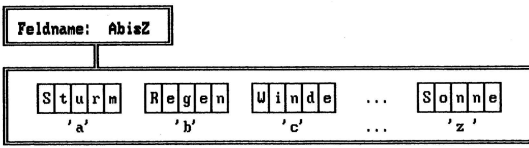
Beispiele:

```
var Monat: array (. 1..31 .) of integer;          (1)
    Werte: array [ -5..4] of real;                (2)
    Reihe: array (. 10..1 .) of integer           (3)
    AbisZ: array ['a'..'z'] of string [5]         (4)
```

Im Beispiel (1) wird ein Feld unter dem Namen Monat vereinbart, das aus 31 INTEGER-Elementen besteht. Die Vereinbarung im Beispiel (2) erzeugt ein Feld mit dem Namen Werte, das 10 REAL-Elemente umfaßt. Das erste Element des Feldes wird mittels Werte (.-5.), das letzte über Werte (.4.) angesprochen.



Die Vereinbarung im Beispiel (3) ist falsch, da der Wert der Untergrenze größer ist als der Wert der Obergrenze.
Im Beispiel (4) schließlich wird ein Feld deklariert, das sich aus 26 Elementen zusammensetzt. Die Elemente bestehen aus 5 Zeichen langen Zeichenketten (String). Das erste Element wird adressiert mittels AbisZ (.'a'.). Die Adresse des letzten Elementes lautet: AbisZ (.'z'.).



2.1.3 Adressierung eines Feldes

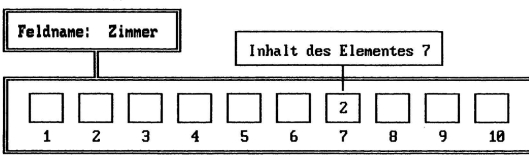
Um ein Element eines Feldes ansprechen bzw. adressieren zu können, müssen der Name des Feldes sowie die Nummer des gewünschten Elements angegeben werden.
Soll beispielsweise vermerkt werden, daß das Zimmer 7 mit zwei Personen belegt wird, so könnte das wie folgt angegeben werden:

```
Zimmer (. 7 .) := 2
```

bzw.

```
Zimmer [7] := 2
```

Wirkung:



In unserem Beispiel trägt das Feld den Namen Zimmer. Die einzelne Elementangabe wird in eckigen Klammern oder in runden Klammern mit Punkt hinter dem Feldnamen vermerkt. Die Elementangabe bezeichnet man auch als **Index** (lat.: Anzeiger).

Der Index kann direkt angegeben werden, beispielsweise durch die Angabe einer ganzzahligen Ziffer. Man spricht dann von **direkter Adressierung** eines Elementes.
Die zweite Möglichkeit besteht darin, daß man in einer Variablen angibt, welches Element des Feldes angesprochen werden soll.

Beispiel:
Beinhaltet die INTEGER-Variable-Nummer den Wert 7, so haben die nachstehenden Anweisungen dieselbe Wirkung:

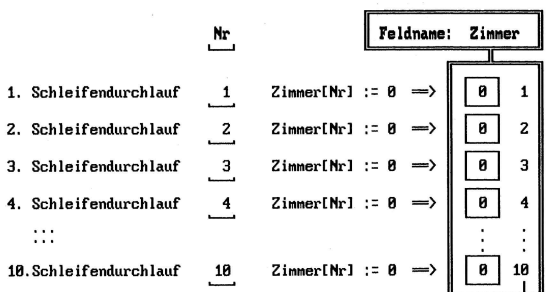
```
Zimmer (.7.) := 2
Zimmer (.Nummer.) := 2
```

Statt der Indexziffer steht nun eine Indexvariable in den Indexklammern, über deren Inhalt ein bestimmtes Element adressiert wird. Schließlich kann der Index auch über einen Ausdruck angegeben werden.

Beispiel: Zimmer (.Nummer + 1.)

Diese Art der Adressierung wird **indizierte Adressierung** genannt.
Die Verwendung von Indexvariablen vereinfacht die Programmierung, wie das folgende Beispiel zeigt. In unserem Hotel muß, wenn alle Gäste abgereist sind, die Zimmerbelegung auf Null gesetzt werden. Mit nur einer Anweisung läßt sich das erreichen:

```
for Nr := 1 to 10 do Zimmer [Nr] := 0;
```



Wie die Abbildung auf Seite 41 zeigt, wird durch die FOR-Anweisung den einzelnen Elementen des Feldes Zimmer der Wert 0 zugewiesen. Das Feld kann – abgesehen von einer Ausnahme, die später erläutert wird – **nicht insgesamt** angesprochen werden. Eine Anweisung der Form

```
Zimmer := 0
```

ist also nicht zulässig. Es muß stets ein Index angegeben werden. Welche Form der Indexangabe der Programmierer wählt, hängt von der Problemstellung ab. Es ist jedoch zu beachten, daß der Datentyp der Indexangabe **ordinal** sein muß. Es dürfen also keine REAL-Zahlen, REAL-Variablen oder REAL-Ausdrücke für Indexangaben verwendet werden. Meistens benutzt man INTEGER-Zahlen, manchmal auch Buchstaben für die Indizierung.

2.1.4 Anwendungsbeispiele

Programmbeispiel **Hotelzimmerbelegung**

Die Kenntnisse über die eindimensionale Datenstruktur werden nun auf das Hotelbeispiel angewendet. Es ist in der Weise zu erweitern, daß die Arbeitsschritte

- Eingabe der Zimmernummer
- Eingabe der Personenzahl
- Zuweisung der Personenzahl auf die Zimmernummer
- Anzeige der aktuellen Zimmerbelegung

so lange wiederholt werden, bis auf eine Ende-Abfrage ein 'j' eingegeben wurde.

Algorithmus (Struktogramm)

```
Algorithmus Zimmerbelegung
Konstante MinNr = 1; MaxNr = 10
Aufbau Bildschirmmaske
Für Nr := MinNr bis MaxNr
    Zimmer [Nr]:= 0
Belegung_Zimmer_anzeigen
    Eingabe Nr
    Eingabe Anzahl
    Zimmer [Nr] := Anzahl
    Belegung_Zimmer_anzeigen
    Eingabe Ende
bis Ende = 'j'
Ende
```

```
Unterprogramm Belegung_Zimmer_anzeigen
Für Nr := MinNr bis MaxNr
    Ausgabe (Nr, Zimmer[Nr])
Ende
```

Programm

```
 0  program Zimmerbelegung;
 1  uses Crt;   (* 4.0 : Diese Anweisung gilt nur für Vers. 4.0*)
 2  const MinNr = 1; MaxNr = 10;
 3  var Nr, Anzahl : integer;
 4      Ende       : char;
 5      Zimmer     : array [MinNr..MaxNr] of integer;


 6  procedure Belegung_Zimmer_anzeigen;
 7  begin
 8    Gotoxy(1,10);
 9    for Nr := MinNr to MaxNr do writeln (Nr:5, Zimmer[Nr]:10);
10  end;

11  begin                                (* HAUPTPROGRAMM *)
12    ClrScr;
13    Gotoxy (3,3); writeln ('Zimmer-Nr  ');
14    Gotoxy (3,4); writeln ('Personen   ');
15    Gotoxy (3,6); writeln ('Ende (j/n) ');
16    Gotoxy (1,8); writeln ('Zimmer  Personen');
17    Gotoxy (1,9); writeln ('----------------');
18    for Nr := MinNr to MaxNr do Zimmer [Nr] := 0;
19    Belegung_Zimmer_anzeigen;
20    repeat
21      Gotoxy (15,3); ClrEol; readln (Nr);
22      Gotoxy (15,4); ClrEol; readln (Anzahl);
23      Zimmer [Nr] := Anzahl;
24      Belegung_Zimmer_anzeigen;
25      Gotoxy (15,6); ClrEol; readln (Ende);
26    until Ende = 'j';
27  end.
```

Erläuterungen

Das Feld Zimmer wurde in der Programmzeile 5 vereinbart. Untergrenze und Obergrenze wurden durch zwei vorher vereinbarte Konstanten – MinNr und MaxNr – festgelegt. Das hat den Vorteil, daß bei einer anderen Zimmeranzahl nur die Konstante MaxNr verändert werden muß.

Wenn Sie mit Turbo-Pascal Version 4.0 arbeiten, müssen Sie mit

```
uses Crt;      (siehe Zeile 1)
```

das Unit Crt aufrufen, in dem die Bildschirmsteuerbefehle behandelt werden. Verwenden Sie die Version 3.x, so dürfen Sie die uses-Anweisung nicht benutzen.

***Programmbeispiel* Kostenaufteilung**

Im folgenden Beispiel wird die Methode der Programmentwicklung, wie sie im Band 1 dargelegt worden ist, stärker als bislang hervorgehoben.

- **Problem**

  Acht Arbeitskollegen haben sich gemeinsam einen Campingbus gekauft. Sie haben vereinbart, daß die anfallenden Kosten (Benzin, Versicherungen usw.) entsprechend den gefahrenen Kilometern aufgeteilt werden.

- **Klärung der Problemstellung**

  Um die entstandenen Kosten auf die einzelnen Benutzer aufteilen zu können, ist zum einen der Kostenbetrag einzugeben. Zum anderen sind die Kilometer, die von den einzelnen Benutzern gefahren worden sind, dem Rechner mitzuteilen. Es sind dann die Kosten pro Kilometer zu berechnen. Den Kostenanteil pro Benutzer erhält man, indem die Kosten pro Kilometer mit den vom Benutzer zurückgelegten Kilometern multipliziert werden.

  Die nachstehende Bildschirmabbildung zeigt das Ergebnis einer solchen Kostenaufteilung:

```
Kosten: 742.55

Name        Kilometer    Kostenanteil
-------------------------------------
Meier             458          182.87
Mueller           251           56.38
Schulz            487          189.38
Otto              265           59.52
Streeb            214           48.87
Paulsen           563          126.45
Hahnich           214           48.87
Summer            854          191.81
-------------------------------------
                 3306          742.55
```

Datenanalyse

		Datentyp
Eingabedaten:	Kosten	REAL
	Namen der Benutzer	STRING (.15.)
	gefahrene Kilometer pro Benutzer	INTEGER
Ausgabedaten:	Summe der gefahrenen Kilometer	INTEGER
	Kostenanteil pro Benutzer	REAL
	Summe der Kostenanteile	REAL

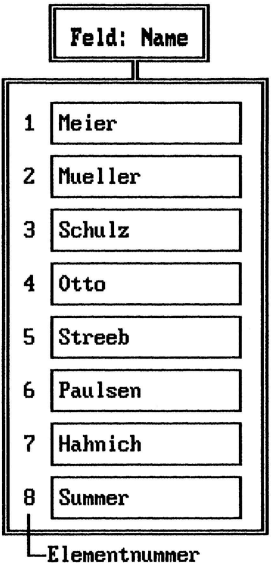
Struktur der Daten:
Die Namen könnten in einen ARRAY of STRING (.15.) eingegeben werden. Da sie aber bei jeder Abrechnung gleich bleiben, ist es günstig, sie als Konstanten zu vereinbaren. Turbo-Pascal bietet hier die Möglichkeit, **typisierte Konstanten** zu verwenden.
Typisierte Konstanten kann man mit Variablen vergleichen, die mit einem Wert vorbelegt werden. Im Gegensatz zu den "normalen", untypisierten Konstanten werden bei der typisierten Konstanten der Datentyp und der Wert vereinbart. Allgemein lautet die Vereinbarung einer typisierten Konstanten:

const Bezeichner: Datentyp = Wert;

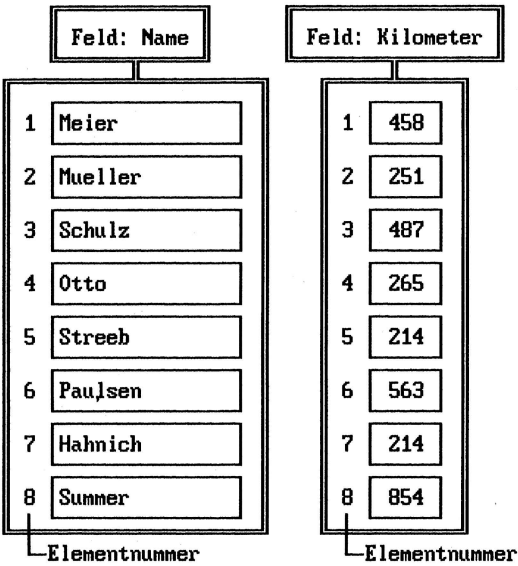
Bezogen auf unser Beispiel können die Namen der Benutzer wie folgt vereinbart werden:

```
const Name: array (.1..8.) of string (.15.) = ('Meier','Mueller',
'Schulz','Otto','Streeb','Paulsen','Hahnich','Summer');
```

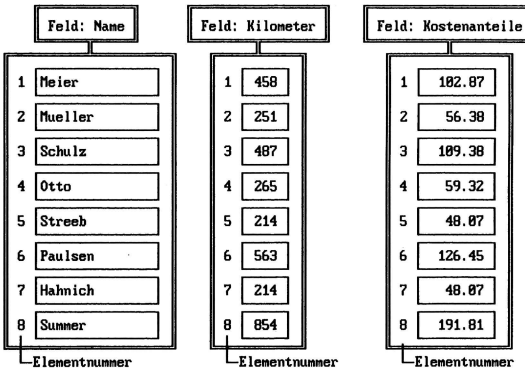
Mit dieser Konstantenvereinbarung wird ein eindimensionales Feld festgelegt, das über 8 Elemente verfügt. Jedes Element besteht aus einem 15 Stellen langen String. In die einzelnen Elemente trägt dann das System die Namen in der angegebenen Reihenfolge ein. Die interne Darstellung im Rechner kann man sich wie folgt vorstellen:



Die Kilometerangaben pro Benutzer müssen einzeln im Rechner gespeichert werden, da sie für mehrere Rechenoperationen benötigt werden. Deshalb ist für sie ein eindimensionales Feld vorzusehen. Die richtige Zuordnung zwischen den Namen und den gefahrenen Kilometern ist sichergestellt, wenn die Kilometerangaben in der Reihenfolge eingegeben werden, in der auch die Namen aufgeführt sind. Die Abbildung im Arbeitsspeicher kann man sich dann wie auf Seite 48 vorstellen:



Auch für den Kostenanteil pro Benutzer wird ein eindimensionales Feld vom Typ REAL eingerichtet.
Die interne Darstellung im Arbeitsspeicher kann man sich dann wie folgt vorstellen:



Beziehungen zwischen Eingabedaten und Ausgabedaten:
Anmerkung: N ist der Index, der die Werte von 1 bis 8 annimmt.
Die Summe der Kilometer ergibt sich aus der Summierung der einzelnen Einträge im Array Kilometer:
SummeKilometer := SummeKilometer + Kilometer (.N.)
Die Kosten pro Kilometer errechnen sich aus der Division von Kosten durch SummeKilometer:
Kilometerkosten := Kosten / SummeKilometer
Die Kostenanteile pro Benutzer bestimmen sich aus der Multiplikation der Kilometerangaben des jeweiligen Benutzers mit den Kosten pro Kilometer:
Kostenanteil (.N.) := Kilometer (.N.) *KilometerKosten
Die Summe der Kostenanteile ergibt aus
SummeKosten := SummeKosten + Kostenanteil (.N.) .
Damit sind alle Ausgabedaten auf Eingabedaten zurückgeführt. Der Algorithmus kann aufgebaut werden (siehe Seite 50).

Algorithmus (Struktogramm)

```
Algorithmus Kostenaufteilung
Konstante  MaxNr = 8
           Name = ('Meier','Mueller','Schulz','Otto','Streeb',
                   'Paulsen','Hahnich','Summer')
Bildschirmmaske aufbauen
Für N := 1 bis MaxNr
    Ausgabe Name (.N.)
    Eingabe Kilometer (.N.)
SummeKilometer := 0
Für N := 1 bis MaxNr
    SummeKilometer := SummeKilometer + Kilometer (.N.)
Kilometerkosten := Kosten / SummeKilometer
Ausgabe SummeKilometer
SummeKosten := 0
Für N := 1 bis MaxNr
    Kostenanteil (.N.) := runde (Kilometerkosten * Kilometer (.N.)
    SummeKosten := SummeKosten + Kostenanteil (.N.)
Für N := 1 bis MaxNr
    Ausgabe Kostenanteil (.N.)
Ausgabe SummeKosten
Ende
```

```
Funktion runde (Wert : REAL) : REAL
runde := int(Wert*100 + 0.500000001)/100
Ende
```

Programm

```
program Kostenaufteilung;
uses Crt;                (* 4.0 *)
const MaxNr = 8;
      Name : array (.1..MaxNr.) of string (.15.)
           = ('Meier', 'Mueller', 'Schulz', 'Otto', 'Streeb',
              'Paulsen', 'Hahnich', 'Summer');
var Kilometer    : array (.1..MaxNr.) of integer;
    Kostenanteil : array (.1..MaxNr.) of real;
    Kosten, Kilometerkosten, SummeKosten : real;
    N,SummeKilometer                     : integer;


function runde (Wert: real): real;
begin
  runde := int(wert*100 + 0.500000001)/100;
end;


begin                                    (* HAUPTPROGRAMM *)
  ClrScr;
  Gotoxy (5,3); write   ('Kosten: '); readln (Kosten);
  Gotoxy (5,5); writeln ('Name       Kilometer   Kostenanteil');
  Gotoxy (5,6); writeln ('-----------------------------------');
  for N := 1 to MaxNr do
  begin
    Gotoxy (5,6+N); writeln (Name(.N.));
    Gotoxy (21,6+N); readln (Kilometer (.N.));
  end;
  Gotoxy (5,15); writeln ('-----------------------------------');
  SummeKilometer := 0;
  for N := 1 to MaxNr do
    SummeKilometer := SummeKilometer + Kilometer(.N.);
  Kilometerkosten := Kosten / SummeKilometer;
  Gotoxy (19,16); writeln (SummeKilometer:6);
  SummeKosten := 0;
  for N := 1 to MaxNr do
  begin
    Kostenanteil (.N.) := runde(Kilometerkosten*Kilometer(.N.));
    SummeKosten := SummeKosten + Kostenanteil (.N.);
  end;
  for N := 1 to MaxNr do
  begin
    Gotoxy (30,6+N);
    writeln (Kostenanteil(.N.):8:2);
  end;
  Gotoxy (30,16); writeln (SummeKosten:8:2);
end.
```

2.1.5 Zusammenfassung

- Ein **Feld (Array)** setzt sich aus einer bestimmten Anzahl von Elementen zusammen.
- Die Elemente müssen den **gleichen Datentyp** haben.
- In der Feldvereinbarung wird durch die Angabe der **Untergrenze** und **Obergrenze** festgelegt, aus wieviel Elementen das Feld bestehen soll. Der Wert der Untergrenze muß kleiner sein als der Wert der Obergrenze. Ferner ist der Datentyp der Elemente zu vermerken.
- Ein Element eines Feldes wird durch den **Feldnamen** und den **Index** adressiert.
- Der Index kann durch
  - eine ordinale Konstante
  - eine Indexvariable, deren Datentyp ordinal ist
  - einen Ausdruck, dessen Wert ordinal ist

  angegeben werden.
- Eine **typisierte Konstante** gleicht einer vordefinierten Variablen.
- Bei der Vereinbarung einer typisierten Konstante wird sowohl der Datentyp als auch der Wert angegeben.

2.1.6 Übungen

1. Aufgabe
Wieviel Elemente umfaßt das Feld TT, das wie folgt vereinbart wurde:

```
var TT: array (. -8..19.) of real;
```

2. Aufgabe
Korrigieren Sie die Feldvereinbarung:

```
var Feld: (. 2...19 .) real;
```

3. Aufgabe
In dem obigen Programmbeispiel Zimmerbelegung soll nach Beendigung der Zimmervergabe zum Tagesabschluß berechnet werden, wie viele Gäste einquartiert sind.
Schreiben Sie eine entsprechende Prozedur!

4. Aufgabe
Folgender Programmausschnitt ist vorgegeben:

```
program MaximumSuche;
const Anzahl = 10;
      Reihe: array (.1..Anzahl.) of integer = (23, 19, 46, 78, 11,
                                               45, 88, 23, 5, 7);
begin
  ...
```

Schreiben Sie das Programm, mit dem festgestellt werden kann, an wievielter Stelle die größte Zahl steht!
Folgender Algorithmus mag Ihnen als Hilfestellung dienen:

Algorithmus MaximumSuche	
Konstante Anzahl = 10 **Reihe = (23,19,46,78,11,45,88,23,5,7)**	
Max := 1	
Für N := 2 bis Anzahl	
Reihe (.N.) > Reihe (.Max.) **ja**	**nein**
Max := N	**./.**
Ausgabe 'Die groesste Zahl steht auf Platz', Max	
Ende	

5. Aufgabe
Gestalten Sie das Programm zu Aufgabe 4 in der Weise um, daß angegeben wird, an wievielter Stelle die kleinste Zahl steht!

2.1.7 Lösungen

1. Aufgabe
Das Feld umfaßt 28 Elemente.

2. Aufgabe
Die korrekte Feldvereinbarung lautet:

```
var Feld: array (. 2..19 .) of real;
```

3. Aufgabe

```
procedure Gaeste_Anzahl;
var Summe : integer;
begin
  Summe := 0;
  for Nr := MinNr to MaxNr do
    Summe := Summe + Zimmer(.Nr.);
  writeln ('Anzahl der Gaeste:',Summe:5);
end;
```

4. Aufgabe

```
program MaximumSuche;
const Anzahl = 10;
      Reihe : array (.1..Anzahl.) of integer
              = (23,19,46,78,11,45,88,23,5,7);
var Max, N  : integer;
begin
  Max := 1;
  for N := 2 to Anzahl do
    if Reihe (.N.) > Reihe (.Max.)
      then Max := N;
  writeln ('Die groesste Zahl steht auf Platz',Max:4);
end.
```

5. Aufgabe

```
program Minimumsuche;
const Anzahl   = 10;
      Reihe    : array (.1..Anzahl.) of integer
                 = (23,19,46,78,11,45,88,23,5,7);
var Minimum, N : integer;
begin
  Minimum := 1;
  for N := 2 to Anzahl do
    if Reihe (.N.) < Reihe (.Minimum.)
      then Minimum := N;
  writeln ('Die kleinste Zahl steht auf Platz',Minimum:4);
end.
```

2.2 Zweidimensionale Felder

Einführungsbeispiel: Ein Hotel vermietet seine 10 Bungalows während der 16wöchigen Sommersaison wochenweise. Damit der Mitarbeiter im Reservierungsbüro einen besseren Überblick über die Bungalowbelegung hat, soll ein Programm eingesetzt werden, das auf dem Bildschirm anzeigt,

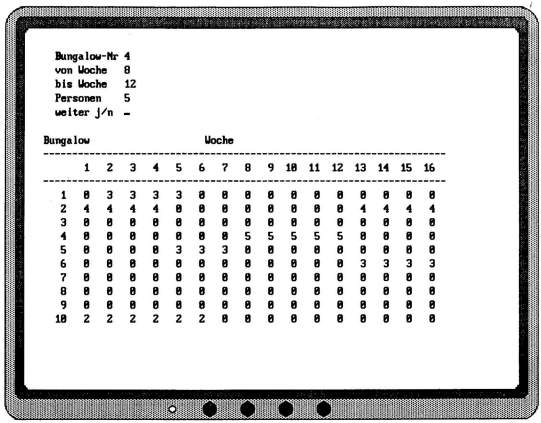
welcher Bungalow
in welchen Wochen
mit wieviel Personen

belegt ist.

Um einen Bungalow zu reservieren, gibt er ein:

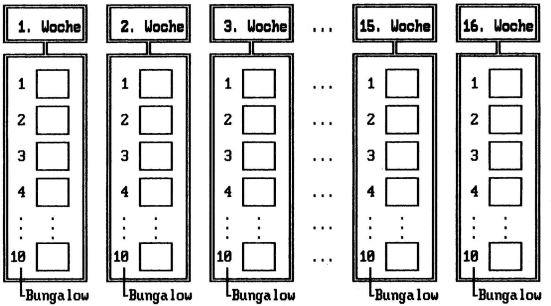
- die Nummer des Bungalows,
- die erste Belegungswoche,
- die letzte Belegungswoche,
- die Anzahl der Personen

Auf dem Bildschirm wird die Belegung wie folgt dargestellt:

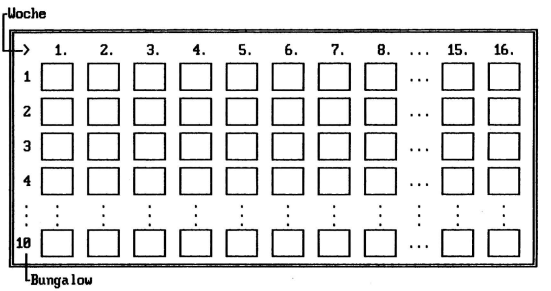


2.2.1 Charakteristik eines zweidimensionalen Feldes

Die im vorigen Kapitel behandelte Datenstruktur des eindimensionalen Feldes würde ausreichen, wenn die Bungalowbelegung nur für jeweils eine Woche interessierte. In dieser Aufgabenstellung muß sie aber für 16 Wochen ausgewiesen werden. Deshalb müßten für den Belegungsplan 16 eindimensionale Felder bereitgestellt werden.



Eine solche Datendarstellung im Rechner würde jedoch zu einer sehr umständlichen Programmierung führen. Deshalb ist es in Pascal wie auch in anderen Programmiersprachen möglich, mehrere eindimensionale Felder zu einem zweidimensionalen Feld zusammenzufassen.



Diese Datenstruktur wird zweidimensional genannt, weil sie sich in zwei Richtungen – vertikal und horizontal – erstreckt. Man kann sich somit das zweidimensionale Feld als ein Gebilde von Zeilen und Spalten vorstellen. Die zweidimensionale Datenstruktur wird deshalb auch als **Matrix** oder **Tabelle** charakterisiert.

Ein zweidimensionales Feld besteht aus einer bestimmten Anzahl von Zeilen und Spalten.

Realisierung im Programm

Wie das zweidimensionale Feld im Programm behandelt wird, zeigt das nachstehende Programm zur Erstellung des Bungalow-Belegungsplanes:

```
1  program Bungalowbelegung;
2  uses Crt;        (* 4 *)
3  const MinNr    = 1; MaxNr    = 10;
4        MinWoche = 1; MaxWoche = 16;
5  var Nr, Anzahl, Woche, AnfWoche, EndWoche : integer;
6      Weiter                              : char;
7      Bungalow : array (.MinWoche .. MaxWoche,MinNr..MaxNr.)
                        of integer;


8  procedure Belegung_Bungalow;
9  begin
10   Gotoxy(1,12);
11   for Nr := MinNr to MaxNr do writeln (Nr:4);
12   for Woche := MinWoche to MaxWoche do
13     for Nr := MinNr to MaxNr do
14     begin
15       Gotoxy (Woche*4+1,Nr+11);
16       write (Bungalow(. Woche,Nr .):4);
17     end;
18 end;


19 procedure Bildschirm;
20 begin
21   ClrScr;
22   Gotoxy (3,2); write ('Bungalow-Nr  ');
23   Gotoxy (3,3); write ('von Woche    ');
24   Gotoxy (3,4); write ('bis Woche    ');
25   Gotoxy (3,5); write ('Personen     ');
26   Gotoxy (3,6); write ('weiter j/n ');
27   Gotoxy (1,8); writeln ('Bungalow                          Woche');
28   Gotoxy (1,9); write ('-----------------------------------');
```

```
29    writeln ('------------------------------');
30    Gotoxy (5,10);
31    for Woche := MinWoche to MaxWoche do write (Woche:4);
32    writeln;
33    write ('-------------------------------------------------------');
34    write ('--------------');
35    for Woche := MinWoche to MaxWoche do
36      for Nr := MinNr to MaxNr do Bungalow (. Woche, Nr .) := 0;
37    Belegung_Bungalow;
38  end;


39  begin  (* HAUPTPROGRAMM *)
40    Bildschirm;
41    repeat
42      Gotoxy (15,2); ClrEol; readln (Nr);
43      Gotoxy (15,3); ClrEol; readln (AnfWoche);
44      Gotoxy (15,4); ClrEol; readln (EndWoche);
45      Gotoxy (15,5); ClrEol; readln (Anzahl);
46      for Woche := AnfWoche to EndWoche do
47        Bungalow (.Woche,Nr.) := Anzahl;
48      Belegung_Bungalow;
49      Gotoxy (15,6); ClrEol; readln (weiter);
50    until weiter = 'n';
51  end.
```

2.2.2 Vereinbarung eines zweidimensionalen Feldes

Die Vereinbarung des zweidimensionalen Feldes wurde in Zeile 7 vorgenommen.

Bungalow: array (.MinWoche..MaxWoche,MinNr..MaxNr.) of integer;

Als Feldname wurde hier "Bungalow" festgelegt. Hinter einem Doppelpunkt wird das Schlüsselwort **array** angegeben. In eckigen Klammern oder in runden Klammern mit Punkt wird die Größe des Feldes vermerkt. Ein wenig vereinfacht kann man sich die Angabe der Feldgröße wie folgt vorstellen:

(. erste Zeile .. letzte Zeile, erste Spalte .. letzte Spalte .)

oder

(. erste Spalte .. letzte Spalte, erste Zeile .. letzte Zeile .)

Es ist also gleichgültig, ob erst die Zeilenangabe und dann die Spaltenangabe gemacht wird. In unserem Beispiel hätte demnach die Feldvereinbarung auch wie folgt vorgenommen werden können:

```
Bungalow: array (.MinNr..MaxNr,MinWoche..MaxWoche.) of integer;
```

Wichtig ist jedoch, daß die **einmal getroffene Vereinbarung** im weiteren Programm **beibehalten** wird. Um sich den Aufbau eines zweidimensionalen Feldes im Arbeitsspeicher besser vorstellen zu können, ist es zweckmäßig, erst die Angaben zu den Spalten und dann die Angaben zu den Zeilen zu machen. In dieser Reihenfolge wird ja auch in der Gotoxy-Anweisung die Spalten- und Zeilenangabe zur Positionierung des Cursors auf dem Bildschirm vorgenommen.
Zeilen- und Spaltenangabe sind durch ein Komma voneinander zu trennen.
Hinter den Feldklammern folgt auf das Schlüsselwort **of** die Angabe des Datentyps der Elemente, aus denen das Feld besteht. Das weist darauf hin, daß nur Daten gleichen Typs zu einem Feld zusammengefaßt werden können.

Die Elemente eines zweidimensionalen Feldes müssen vom gleichen Datentyp sein.

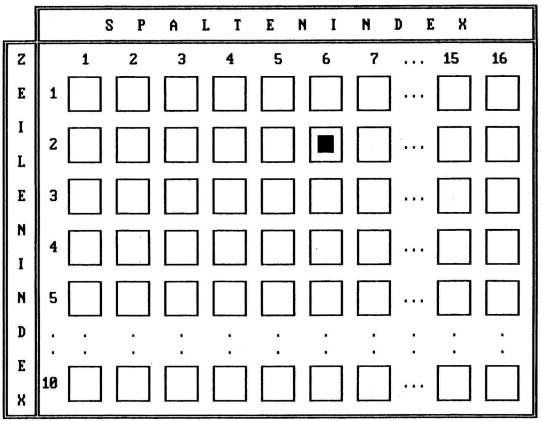
Allgemein lautet die Vereinbarung eines zweidimensionalen Feldes:

var Name: array (. UG1 .. OG1,UG2 .. OG2 .) of Datentyp;

UG (= Untergrenze) und OG (= Obergrenze) sind beliebige Konstanten ordinalen Typs. In der Regel werden INTEGER-Konstanten oder Zeichen-Konstanten verwendet, um die Größe des Feldes festzulegen. Der Wert für UG muß stets kleiner sein als der Wert für OG.

2.2.3 Adressierung eines zweidimensionalen Feldes

Zur Adressierung eines Elements in einem zweidimensionalen Feld müssen **zwei Indizes** (Mehrzahl von Index) angegeben werden. Der eine Index bezeichnet die Spalte, der andere die Zeile, in der sich das gewünschte Element befindet.



Um auf das mit ■ gekennzeichnete Element zugreifen zu können, müßte angegeben werden:
(1) ■ Feldname (2,6) oder
(2) ■ Feldname (6,2).

Ob die Adressierung in (1) oder (2) richtig ist, hängt davon ab, wie die Matrix vereinbart wurde.
Die Indizes können direkt über Konstanten angegeben werden. Es können aber auch Variablen oder Ausdrücke ordinalen Datentyps als Indexangaben verwendet werden.
In unserem Programmbeispiel Bungalowbelegung wird der Zugriff auf ein Element des zweidimensionalen Feldes über zwei Indexvariablen vorgenommen, wie Sie aus Zeile 16 (Seite 57) des Programms ersehen können.

```
16  write (Bungalow(. Woche,Nr .):4);
```

Die Indexvariablen lauten Woche und Nr. In dieser Reihenfolge müssen sie angegeben werden, weil in der Feldvereinbarung zuerst der Index für die Wochen-Nummer und anschließend der Index für die Bungalow-Nummer vereinbart wurde.
Die Reihenfolge der Indexangaben muß mit der Feldvereinbarung übereinstimmen. Sonst kommt es zu einem Laufzeitfehler infolge einer Bereichsüberschreitung.

Bereichsüberprüfung
Probieren Sie es einmal selber: Geben Sie das obige Programm ein und speichern Sie es ab! Anschließend verändern Sie in der Anweisung

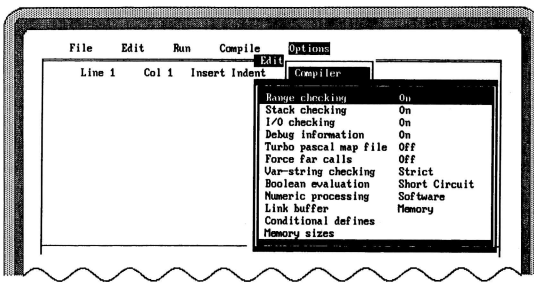
```
16  write (Bungalow(. Woche,Nr .):4);
```

in der Weise, daß Sie die Indexangaben vertauschen:

```
16  write (Bungalow(. Nr,Woche .):4);
```

Wenn Sie nun das Programm starten, werden möglicherweise unsinnige Werte am Bildschirm angezeigt. Das liegt daran, daß auf Speicherbereiche zugegriffen wird, die nicht im vereinbarten Bereich liegen. (Speichern Sie das fehlerhafte Programm bitte nicht ab!)
Solche Laufzeitfehler sind nicht nur ärgerlich, sondern auch gefährlich! Nicht immer erkennt man sofort den Zugriffsfehler.
Damit der Rechner einen solchen Zugriffsfehler feststellen kann, muß man das Programm um den Compiler-Befehl (*$R+*) erweitern. Man setzt diesen Befehl vor den Vereinbarungsteil des Programms. Dadurch wird ein "Inspektor" im Turbo-Pascal-System aktiviert, der die Zugriffe auf ein Feld überprüft. Überschreitet eine Indexangabe die Feldgrenze, bricht er das Programm mit einer Fehlermeldung (Error 201) ab.
Ergänzen Sie das Programm Bungalowbelegung um den Compiler-Befehl (*$R+*)! Starten Sie das Programm, und geben Sie unzulässige Werte ein! Der Befehl (*$R+*) bewirkt, daß der Programmlauf mit einer Fehlermeldung gestoppt wird.
Wenn Sie mit der Version 4.0 arbeiten, so können Sie die Bereichsüberprüfung auch über das Menü **Options** einstellen. Rufen Sie es mit <ALT> <O> auf! Wählen Sie durch Betätigen der RETURN-Taste die Compiler-Optionen an! Der Balken steht auf Range checking (Bereichsüberprüfung). Es weist in der Regel die Einstellung off auf.

Drücken Sie die RETURN-Taste! Der Prüfschalter ist nun auf On gesetzt. Durch zweimaliges Betätigen der Taste <ESC> und anschließendes <ALT> <R> starten Sie das Programm! Nach der Eingabe von unzulässigen Werten können Sie nun die Wirkung der automatischen Bereichsüberprüfung feststellen.



2.2.4 Schleifenkonstruktionen bei Tabellenverarbeitung

Bei zweidimensionalen Feldern oder Tabellen gibt es einige Standard-Verarbeitungsschritte:

- Einlesen von Werten in die Tabelle,
- Zuweisung von Werten an die Tabelle,
- Summierung von Tabellenwerten,
- Suchen von Werten in der Tabelle.

Bei diesen Aufgaben müssen in der Regel alle Elemente der Tabelle angesprochen werden. Da die Elemente aber nur über Indizes adressiert werden können, verwendet man Schleifenkonstruktionen, in denen die erforderlichen Indexangaben gebildet werden.
Bei dieser Schleifenbildung sind zwei Grundformen zu unterscheiden:

```
for Zeile := ErsteZeile to LetzteZeile do
    for Spalte := ErsteSpalte to LetzteSpalte do
    Anweisung;
```

oder

```
for Spalte := ErsteSpalte to LetzteSpalte do
    for Zeile := ErsteZeile to LetzteZeile do
    Anweisung;
```

Bei einigen Problemstellungen ist es gleichgültig, in welcher Reihenfolge die beiden Schleifen angeordnet sind, welche der beiden Schleifen also übergeordnet bzw. untergeordnet ist. Es gibt aber auch Aufgabenstellungen, bei denen es sehr wohl darauf ankommt, daß die Zeilen-Schleife bzw. die Spalten-Schleife übergeordnet ist.

Reihenfolge nicht entscheidend

Ein Beispiel dafür, daß die Reihenfolge der Zeilen- bzw. Spalten-Schleifen nicht für die Korrektheit der Programmschritte entscheidend ist, ist im Programm Bungalow (Seite 57) in den Zeilen 35 und 36 zu finden.

```
35  for Woche := MinWoche to MaxWoche do
36  for Nr := MinNr to MaxNr do Bungalow (. Woche,Nr .) := 0;
```

In dieser Schleifenkonstruktion wird den einzelnen Elementen der Tabelle der Wert 0 zugewiesen. Wenn Sie das Programm eingegeben und gestartet haben, so haben Sie am Bildschirm sehen können, daß die einzelnen Elemente den Wert 0 aufweisen.
Wenn die Schleifenkonstruktion umgestellt wird, erhalten ebenfalls alle Elemente den Wert 0.

```
35  for Nr := MinNr to MaxNr do
36  for Woche := MinWoche to MaxWoche do
    Bungalow (. Woche,Nr .) := 0;
```

Nehmen Sie im Programm die entsprechende Änderung vor und testen Sie es! Falls Sie keinen Rechner zur Verfügung haben, können Sie sich die Wirkung dieser Schleifenkonstruktion an Hand der Wertbelegungstabelle auf Seite 64 verdeutlichen:

Nr.	Woche	Wertzuweisung auf
1	1	Bungalow (.1,1.)
1	2	Bungalow (.2,1.)
1	3	Bungalow (.3,1.)
..	..	..
1	16	Bungalow (.16,1.)
2	1	Bungalow (.1,2.)
2	2	Bungalow (.2,2.)
2	3	Bungalow (.3,2.)
..	..	..
2	16	Bungalow (.16,2.)
3	1	Bungalow (.1,3.)
..	..	..
10	15	Bungalow (.15,10.)
10	16	Bungalow (.16,10.)

Wie aus der Wertbelegungstabelle zu ersehen ist, werden alle Elemente des Feldes in die Wertzuweisung einbezogen.
Bei einer Schleifenkonstruktion wie

```
for Nr := MinNr to MaxNr do
for Woche := MinWoche to MaxWoche do
    Bungalow (. Woche,Nr .): = 0;
```

besteht nämlich ein geordnetes Verhältnis. Die Nr-Schleife ist der Woche-Schleife übergeordnet. Zu Beginn der Schleifenbearbeitung wird der Variablen Nr der Anfangswert zugewiesen. Diesen Wert behält Nr – wie aus der Wertbelegungstabelle zu ersehen ist – so lange bei, bis in der untergeordneten Schleife die Variable Woche einen Wert erreicht hat, der nicht mehr kleiner/gleich MaxWoche ist. Erst nach dem Verlassen der Woche-Schleife wird der Wert der Variablen Nr um 1 erhöht. Dieser Wert bleibt nun so lange in Nr erhalten, bis erneut die Woche-Schleife abgearbeitet ist. Der Prozeß wird so lange durchgeführt, bis in der übergeordneten Schleife die Variable Nr nicht mehr kleiner oder gleich dem Wert von MaxNr ist.
Auch in der Schleifenkonstruktion in der Prozedur Bungalow-Belegung ist die Reihenfolge der Woche- und der Nr-Schleife nicht entscheidend.

```
12  for Woche := MinWoche to MaxWoche do
13  for Nr := MinNr to MaxNr do
14    begin
15    Gotoxy (Woche*4+1,Nr+11);
16    write (Bungalow(. Woche,Nr .):4);
17    end;
```

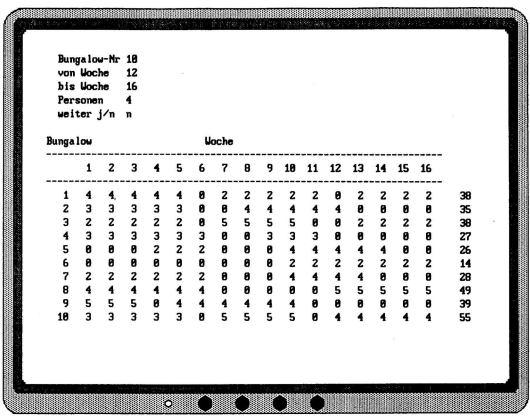
Bei dieser Schleifenkonstruktion wird das Bild zeilenweise erzeugt. Wenn Sie die beiden FOR-Anweisungen vertauschen, erfolgt die Ausgabe spaltenweise. In jedem Fall wird dasselbe Bild auf dem Bildschirm erzeugt. Das ist darauf zurückzuführen, daß über die Gotoxy-Anweisung

```
15    Gotoxy (Woche*4+1,Nr+11);
```

für jede Wertausgabe der Ausgabeplatz auf dem Bildschirm berechnet wird.

Reihenfolge entscheidend

Das obige Programm soll um eine Prozedur erweitert werden, mit der die Anzahl der Personen ermittelt wird, die während der 16 Wochen in den verschiedenen Bungalows wohnen.



Bei dieser Aufgabenstellung handelt es sich um eine Summierungsaufgabe, bei der zeilenweise die Summe ermittelt und ausgegeben wird. Bei dieser Aufgabenstellung ist es sehr wohl entscheidend, welche Schleife übergeordnet und welche untergeordnet ist.
Der Aufbau der Schleife wird nun näher betrachtet. Die Summe der Personen, die in einem Bungalow während der 16 Wochen wohnen, wird ermittelt, indem für jeweils einen Bungalow die 16 Wochenwerte addiert werden. Bezogen auf den Bungalow mit der Nummer 1 würde die FOR-Anweisung lauten:

```
PersonenSum := 0;
for  Woche := 1 to 16 do
     PersonenSum := PersonenSum + Bungalow (.Woche, 1.);
writeln (PersonenSum:4);
```

Für den 2. Bungalow müßte die Anweisungen zur Summierung lauten:

```
PersonenSum := 0;
for Woche := 1 to 16 do
PersonenSum := PersonenSum + Bungalow (.Woche, 2.);
writeln (PersonenSum:4);
```

Aus diesen Anweisungen zur zeilenweisen Summierung läßt sich bereits ableiten, daß die Woche-Schleife untergeordnet sein muß. Sie muß eingeschachtelt sein in die Nr-Schleife, die um die obige Anweisungsfolge herumgelegt wird. Das zeigt auch die nachstehend dargestellte Prozedur Personensumme.

```
procedure Personensumme;
var PersonenSum : integer;
begin
  for Nr := MinNr to MaxNr do
  begin
    PersonenSum := 0;
    for Woche := MinWoche to MaxWoche do
      PersonenSum := PersonenSum + Bungalow (.Woche,Nr.);
    Gotoxy (72,Nr+11);
    write (PersonenSum:4);
  end;
end;
```

In dieser Prozedur wird die Zeilensumme jeweils ab der 72. Spalte auf dem Bildschirm angezeigt.

Ausgabe einer Tabelle über Drucker

Die Ausgabe einer Tabelle wurde bereits behandelt. Da sie über den Bildschirm unter Einbeziehung einer Bildschirmsteuerung erfolgte, war bei ihr die Reihenfolge der FOR-Anweisungen unerheblich. Auf dem Bildschirm kann man direkt bestimmte Stellen ansprechen. Bei der Ausgabe über einen Drucker ist das jedoch nicht möglich. In der Regel arbeiten Drucker nämlich zeilenorientiert. Ein Wert kann in eine Zeile nicht nachträglich eingefügt werden. Deshalb müssen alle in einer Zeile zu druckenden Angaben in der richtigen Reihenfolge mit einer WRITE-Anweisung ausgegeben werden, bevor mittels WRITELN ein Zeilenvorschub veranlaßt wird.
Um eine Tabellenzeile der Bungalow-Belegung auszudrucken, müßte folgende Anweisungsfolge programmiert werden:

```
write (lst,Nr:4,' ');
write (lst,Bungalow(.1,Nr.):4); write (lst,Bungalow(.2,Nr.):4);
write (lst,Bungalow(.3,Nr.):4);... write (lst,Bungalow(.16,Nr.):4);
writeln;
```

Diese Folge von WRITE-Anweisungen, die sich auf das Feld Bungalow beziehen, kann man in einer FOR-Anweisung zusammenfassen:

```
write (lst,Nr:4,' ')
for Woche := MinWoche to MaxWoche do
write (lst,Bungalow (.Woche,Nr.):4);
writeln(lst);
```

Damit der Ausdruck für alle Bungalows vorgenommen wird, ist um diese Anweisungsfolge noch eine Nr-Schleife herumzulegen, wie es in der umseitigen Prozedur Druck-Tabelle dargelegt ist.

```
procedure Druck_Tabelle;
begin
  for Nr := MinNr to MaxNr do
  begin
    write (lst,Nr:4,' ');
    for Woche := MinWoche to MaxWoche do
      write (lst,Bungalow (.Woche,Nr.):4);
    writeln(lst);
  end;
end;
```

Anmerkungen zur Prozedur: Wenn Sie mit der Version 4.0 von Turbo-Pascal arbeiten, müssen Sie das Unit **Printer** in Ihr Programm einbinden. Haben Sie bereits den Aufruf von Unit Crt vermerkt, so setzen Sie den Unit-Namen Printer, abgetrennt durch ein Komma, hinter Crt.

Beispiel: program Bungalow;
uses Crt,Printer;

Gestaltung von Eingaberoutinen

Das obige Programm Bungalowbelegung kann in einer Hinsicht nicht zufriedenstellen: es fängt keine **Eingabefehler** ab. Wenn Sie aus Versehen für Bungalow-Nr eine 11 eingeben, stürzt das Programm ab, weil 11 größer als MaxNr ist. Alle bisherigen Eingaben gehen verloren. Das darf nicht sein!

Um das Programm absturzsicher zu machen, muß jede Eingabe auf Zulässigkeit geprüft werden. Empfehlenswert ist, die Eingaben über eine Eingabeprozedur vorzunehmen. Das ist in diesem Programmbeispiel leicht möglich, da die Eingaben zu

Nr
AnfWoche
EndWoche
Anzahl

stets ganzzahlig sind.

Die Eingabeprozedur wird wie folgt formuliert:

```
procedure Eingabe_Ganzzahl (Spalte,Zeile,UW,OW : integer;
                            var Zahl           : integer);
var Taste : char;
begin
  repeat
    Gotoxy (Spalte,Zeile); ClrEol;
    read (Zahl);
    if (Zahl<UW) or (Zahl>OW)
      then begin
        Gotoxy (Spalte+10,Zeile);
        write ('Eingabe falsch! <Return> drücken');
        Taste := ReadKey (*4*)  (*3: read (kbd, Taste); *)
      end;
  until (Zahl>= UW) and (Zahl<= OW);
end;
```

Erläuterungen

Im obigen Programm wurde die Eingabe der Bungalow-Nummer mit der Anweisungsfolge

42 Gotoxy(15,2); ClrEol; readln (Nr);

vorgenommen. Unter Verwendung der Prozedur Eingabe_Ganzzahl lautet der Prozeduraufruf zur Eingabe der Bungalow-Nummer wie folgt:

42 Eingabe_Ganzzahl (15,2,MinNr,MaxNr,Nr);

In der aktuellen Parameterliste wird der Prozedur in den ersten beiden Elementen (15 und 2) die Cursorpositionierung mitgeteilt. Darauf folgt die Angabe der zulässigen Werte. Hier werden die Werte der Konstanten MinNr und MaxNr an die formalen Parameter UW und OW übertragen. Schließlich wird die Variable Nr an die Prozedur übergeben.

Der Kernpunkt der Prozedur Eingabe_Ganzzahl besteht darin, daß eine Eingabe so lange wiederholt werden muß, bis der eingegebene Wert innerhalb der zulässigen Werte liegt. Falls der Eingabewert unzulässig ist, wird in der Eingabezeile die Warnung angezeigt, daß eine falsche Eingabe vorliegt. Diese Meldung soll der Benutzer mittels

RETURN-Taste bestätigen. Er könnte auch jede andere Taste betätigen. Die Anweisung

```
(*4*)   Taste := ReadKey;
(*3*)   read (kbd, Taste);   (*kbd = keyboard = Tastatur*)
```

bewirkt nämlich, daß das Programm auf eine Tastatureingabe wartet. Nach Betätigen einer Taste fährt das Programm fort. Die Eingabe selber wird nicht auf dem Bildschirm angezeigt, sondern nur in der Variablen Taste gespeichert.
Wenn Sie mit der Version 4.0 von Turbo-Pascal arbeiten, können Sie ebenfalls read(kbd, Taste), benutzen. Dann müssen Sie jedoch hinter uses das Unit Turbo3 anführen.

2.2.5 Zusammenfassung

- Ein **zweidimensionales Feld** besteht aus Elementen gleichen Datentyps, die zeilen- und spaltenweise angeordnet sind.
- Ein zweidimensionales Feld wird auch **Matrix** oder **Tabelle** genannt.
- Ein zweidimensionales Feld wird vereinbart durch
  var Feldname: **array** (.UG1 .. OG1,UG2 ..OG2.) of Datentyp
  UG (= Untergrenze) und OG (= Obergrenze) müssen ordinale Konstanten sein.
- Um sich den Aufbau eines zweidimensionalen Feldes im Arbeitsspeicher besser vorstellen zu können, ist es sinnvoll, erst die Spaltenanzahl und dann die Zeilenanzahl anzugeben.
- Ein Element eines zweidimensionalen Feldes wird adressiert über

  Feldname (.Index1, Index2.)

  Index1 und Index2 müssen **ordinalen Typs** sein.
- Um zu verhindern, daß ein Zugriff über die Grenzen eines Feldes erfolgt, kann in Turbo-Pascal mit dem Compilerbefehl **(*$R+*)** die **Bereichsüberprüfung eingeschaltet** werden. Mit **(*$R−*)** wird sie wieder **ausgeschaltet**.
  Dieser Compilerbefehl kann in der Version 4.0 auch über das Menü Options / Compiler gegeben werden.
- Zur Verarbeitung eines zweidimensionalen Feldes insgesamt werden zwei Formen von geschachtelten Schleifenkonstruktionen benötigt.

1. Form

```
for Zeile := ErsteZeile to LetzteZeile do
    for Spalte := ErsteSpalte to LetzteSpalte do
    Anweisung;
```

2. Form

```
for Spalte := ErsteSpalte to LetzteSpalte do
    for Zeile := ErsteZeile to LetzteZeile do
    Anweisung;
```

- Bei einer zeilenweise orientierten Bearbeitung ist die erste Schleifenform anzuwenden.
- Bei einer spaltenweise orientierten Bearbeitung ist die zweite Schleifenform einzusetzen.
- Um Programmabstürze infolge fehlerhafter Eingaben zu vermeiden, ist mittels einer geeigneten Eingabeprozedur die Zulässigkeit der eingegebenen Werte zu überprüfen.

2.2.6 Übungen

1. Aufgabe
Was ist an folgender Feldvereinbarung falsch?

```
var Werte: array (.1..10,8..1.) of real;
```

2. Aufgabe
Die Zeugnisnoten aller 26 Schüler einer Klasse, die Unterricht in 9 Fächern erhält, sollen in einer Notenmatrix dargestellt werden. Wie ist die Notenmatrix zu vereinbaren?

3. Aufgabe
Das obige Programm Bungalow ist um eine Prozedur zu erweitern, mit der die Gesamtanzahl der Personen, die für die Bungalows angemeldet worden sind, ermittelt wird. Die Gesamtanzahl ist in der 24. Zeile des Bildschirms anzuzeigen. Folgender Algorithmus auf Seite 72 mag Ihnen als Hilfestellung dienen:

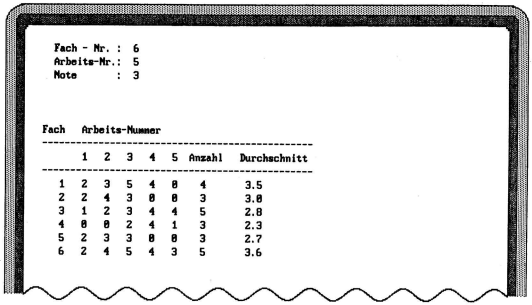
```
Prozedur Gesamtsummierung
PersSumme := 0
Für Nr := MinNr bis MaxNr
    Für Woche := MinWoche bis MaxWoche
        PersSumme := PersonenSumme + Bungalow(.Woche,Nr.)
Ausgabe PersSumme
Ende
```

4. Aufgabe

Die Hotelleitung möchte wissen, wie viele Personen für die verschiedenen Wochen angemeldet worden sind. Schreiben Sie die Prozedur, mit der eine spaltenweise Summierung erreicht wird! Die Summenwerte sind in der 23. Zeile ab der 5. Spalte anzuzeigen.

5. Aufgabe

In einer Klasse werden in einem Schulhalbjahr in 6 Fächern bis zu 5 Klassenarbeiten geschrieben. Es ist ein Programm zu entwickeln, mit dem ein Schüler seine je Fach erzielten Noten eingeben kann. Hat er eine Arbeit nicht mitgeschrieben, gibt er eine Null ein. Sind in einem Fach weniger als 5 Arbeiten geschrieben worden, wird bei den restlichen Notenangaben eine Null eingetippt.
Der Rechner soll ermitteln, wie viele Arbeiten der Schüler geschrieben hat. Ferner soll der Rechner den Notendurchschnitt pro Fach berechnen.



```
Fach - Nr. : 6
Arbeits-Nr.: 5
Note       : 3


Fach   Arbeits-Nummer
-------------------------------------------
       1  2  3  4  5  Anzahl   Durchschnitt
-------------------------------------------
  1    2  3  5  4  0     4        3.5
  2    2  4  3  0  0     3        3.0
  3    1  2  3  4  4     5        2.8
  4    0  0  2  4  1     3        2.3
  5    2  3  3  0  0     3        2.7
  6    2  4  5  4  3     5        3.6
```

2.2.7 Lösungen

1. Aufgabe

Bei der Feldvereinbarung muß der Wert der Untergrenze kleiner sein als der Wert der Obergrenze. Richtig lautet die Vereinbarung:

```
var Werte: array (.1..10,1..8.) of real;
```

2. Aufgabe

Die Vereinbarung der Notenmatrix kann lauten:

```
var Noten: array (.1..9,1..26.) of integer;
```

oder

```
var Noten: array (.1..26,1..9.) of integer;
```

3. Aufgabe

```
procedure Gesamtsummierung;
var PersSumme : integer;
begin
  PersSumme := 0;
  for Mr := MinMr to MaxMr do
    for Woche := MinWoche to MaxWoche do
      PersSumme := PersSumme + Bungalow (.Woche, Mr.);
  Gotoxy (10,24);
  write ('Gesamtanzahl der Personen: ',PersSumme:4);
end;
```

4. Aufgabe

```
procedure SpaltenSummierung;
var WochSum : integer;
begin
  Gotoxy (5,23);
  for Woche := MinWoche to MaxWoche do
  begin
    WochSum := 0;
    for Mr := MinMr to MaxMr do
      WochSum := WochSum + Bungalow (.Woche,Mr.);
    write (WochSum:4);
  end;
end;
```

5. Aufgabe

```
program Noten_Durchschnitt_Berechnung;
(*$R+*)
uses Crt,turbo3;   (* 4 *)
const MinNr   = 1; MaxNr   = 5;
      MinFach = 1; MaxFach = 6;
var Nr, Fach        : integer;
    Noten           : array (.MinNr..MaxNr,MinFach..MaxFach.)
                      of integer;
    Anzahl_Arbeiten : array (. MinFach..MaxFach.) of integer;


procedure Bildschirm;
begin
  ClrScr;
  Gotoxy (3,2); write ('Fach - Nr. : ');
  Gotoxy (3,3); write ('Arbeits-Nr.: ');
  Gotoxy (3,4); write ('Note       : ');
  Gotoxy (1,8); writeln ('Fach   Arbeits-Nummer');
  Gotoxy (1,9);
  write ('-------------------------------------------------');
  Gotoxy (5,10);
  for Nr := MinNr to MaxNr do write (Nr:4);
  writeln ('  Anzahl   Durchschnitt');
  writeln ('-------------------------------------------------');
  for Fach := MinFach to MaxFach do writeln (Fach:4);
end;


procedure Eingabe_Ganzzahl (Spalte,Zeile,UW,OW : integer;
                            var Zahl           : integer);
var Taste : char;
begin
  repeat
    Gotoxy (Spalte,Zeile); ClrEol;
    read (Zahl);
    if (Zahl<UW) or (Zahl>OW)
      then begin
        Gotoxy (Spalte+10,Zeile);
        write ('Eingabe falsch! <Return> drücken');
        read (kbd, Taste);
      end;
  until (Zahl>= UW) and (Zahl<= OW);
end;


procedure Eingabe_Anzeige;
var Weiter : char;
begin
  for Fach := MinFach to Maxfach do
  begin
```

```
    Gotoxy (15,2); write (Fach:3);
    for Nr := MinNr to MaxNr do
    begin
      Gotoxy (15,3); write (Nr:3);
      Eingabe_Ganzzahl(17,4,0,6,Noten(.Nr,Fach.));
      Gotoxy (Nr*4 +1, (Fach-1)*12);
      writeln (Noten (.Nr,Fach.):4);
    end;
  end;
end;


procedure Anzahl_Arbeiten_ermitteln_anzeigen;
begin
  for Fach := MinFach to MaxFach do
  begin
    Anzahl_Arbeiten (.Fach.) := 0;
    for Nr := MinNr to MaxNr do
      if Noten(.Nr,Fach.) > 0
        then Anzahl_Arbeiten(.Fach.) := Anzahl_Arbeiten(.Fach.) +1;
  end;
  for Fach := MinFach to MaxFach do
  begin
    Gotoxy (26,Fach+11);
    write (Anzahl_Arbeiten(.Fach.):4);
  end;
end;


procedure Durchschnitt_berechnen_anzeigen;
var Summe        : integer;
    Durchschnitt : real;
begin
  for Fach := MinFach to MaxFach do
  begin
    Summe := 0;
    for Nr := MinNr to MaxNr do
      Summe := Summe + Noten(.Nr,Fach.);
    Durchschnitt := Summe /Anzahl_Arbeiten(.Fach.);
    Gotoxy (36,Fach+11); write (Durchschnitt:4:1);
  end;
end;


begin                                   (* HAUPTPROGRAMM *)
  Bildschirm;
  Eingabe_Anzeige;
  Anzahl_Arbeiten_ermitteln_anzeigen;
  Durchschnitt_berechnen_anzeigen;
end.
```

3 DATENSTRUKTUR MENGE (SET)

Einführungsbeispiel: Susanne bucht am PC. Plötzlich schimpft sie: "Verflixt, jetzt ist das Programm abgestürzt, weil ich eine falsche Kennziffer eingegeben habe. Statt einer 1,3,5 oder 7 hatte ich eine 4 eingetippt."
Programme müssen unbedingt gegen Eingabefehler abgesichert werden. Es darf nicht sein, daß es auf Grund einer falschen Eingabe zu einem Programmabsturz kommt und somit viele Daten verlorengehen. Dieses Problem wurde bereits kurz im vorigen Kapitel angesprochen.
Mit unseren bisherigen Kenntnissen können wir die Überprüfung, ob ein eingegebener Wert zulässig ist, nur über zusammengesetzte Bedingungsausdrücke programmieren.

Beispiel: Bei einer Eingabe von ganzzahligen Werten sind nur die Ziffern 1, 3, 5 und 7 zulässig. Werden andere Werte eingegeben, führt das zu einem Programmabsturz.

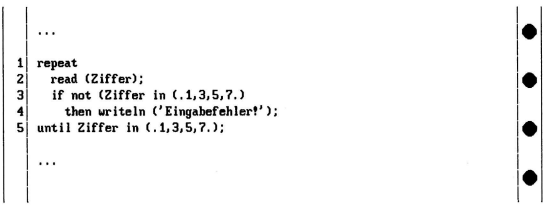
Unter Verwendung von zusammengesetzten Bedingungsausdrücken müßten die Anweisungen zur Werteingabe wie folgt lauten:

```
  ...
repeat
  read (Ziffer);
  if not ((Ziffer=1) or (Ziffer=3) or (Ziffer=5) or (Ziffer=7))
    then writeln ('Eingabefehler!');
until (Ziffer=1) or (Ziffer=3) or (Ziffer=5) or (Ziffer=7);
  ...
```

Die Prüfung von Werten auf Zulässigkeit läßt sich unter Verwendung der Datenstruktur **Menge (SET)** vereinfachen.

> Eine Menge besteht aus einer bestimmten Anzahl von Elementen eines ordinalen, d. h. abzählbaren Datentyps.

Unter Verwendung der Datenstruktur Menge lautet die obige Eingabeprozedur:



```
   ...

1  repeat
2    read (Ziffer);
3    if not (Ziffer in (.1,3,5,7.)
4      then writeln ('Eingabefehler!');
5  until Ziffer in (.1,3,5,7.);

   ...
```

In dieser Eingabeschleife wird in Zeile 3 und 5 geprüft, ob sich der in die Variable Ziffer eingegebene Wert in der Menge der Zahlen (.1,3,5,7.) befindet. Wurde beispielsweise der Wert 2 eingetastet, so stellt das Programm in der Zeile 3 fest, daß dieser nicht in der Menge (.1,3,5,7.) enthalten ist und gibt daraufhin eine Fehlermeldung aus. Aus Zeile 5 ist zu ersehen, daß die REPEAT-Schleife nur dann verlassen wird, wenn sich der eingegebene Wert in der Menge (.1,3,5,7.) befindet.
Der Vergleich der beiden Eingabeschleifen zeigt, daß die Programmierung durch Anwendung der Datenstruktur **Menge** vereinfacht wird.

3.1 Vereinbarung von Mengen

Mengen können nicht direkt über die Tastatur eingegeben werden. Sie werden vielmehr per Anweisungen gebildet.
Mengen können auf verschiedene Weisen in einem Programm erzeugt werden. Die erste Möglichkeit besteht darin, die Menge unmittelbar

anzugeben. Die Elemente einer Menge stehen zwischen eckigen oder runden Klammern mit Punkt. Besteht die Menge aus einem zusammenhängenden Bereich, so genügt es, das erste und das letzte Element des Bereiches anzugeben.

Beispiele:

(1) Sollen die Ziffern von 5 bis 15 zu einer Menge zusammengefaßt werden, so kann die Menge mittels

(.5..15.)

angegeben werden. Zwischen den beiden Werten stehen wie beim Unterbereich zwei (!) Punkte.

(2) Ist eine Menge aus den Kleinbuchstaben zu bilden, so formuliert man

(.'a'..'z'.).

Besteht eine Menge aus Elementen, die sich nicht in einer Reihenfolge befinden, so sind die Elemente einzeln anzugeben.

Beispiel: Die Menge der Vokale formuliert man mittels

(. 'A','E','I','O', 'U', 'a','e','i','o','u'.)

Beide Darstellungsformen können auch gemischt angewendet werden.

Beispiel: (. 0..9,15,30.)

Die zweite Möglichkeit, die Datenstruktur Menge in ein Programm einzubeziehen, besteht in der Vereinbarung einer Mengenvariablen, der eine Menge zugewiesen wird. Das wird am Beispiel der Prozedur Eingabe_Ganzzahl dargestellt (siehe Seite 69). Dort wurde die fehlerhafte Eingabe mittels zusammengesetzter Bedingungsausdrücke festgestellt. In der nachfolgenden Prozedur wird die Fehlerabfrage mit Hilfe einer Mengenvariablen durchgeführt.

```
1  procedure Eingabe_Ganzzahl (Spalte,Zeile,UU,OU : integer;
                               var Zahl           : integer);
2  var Taste          : char;
3      Zulaessige_Zahl : set of 0..100;
4  begin
5    Zulaessige_Zahl := [UU..OU];
6    repeat
7      Gotoxy (Spalte,Zeile); ClrEol;
8      read (Zahl);
9      if not (Zahl in Zulaessige_Zahl)
10       then begin
11         Gotoxy (Spalte+3,Zeile);
12         write ('Eingabe falsch! <Return> drücken');
13         read (Taste);
14       end;
15   until Zahl in Zulaessige_Zahl;
16 end;
```

Bauen Sie die Eingabeprozedur beispielsweise in das Programm Noten_Durchschnitt_Berechnung (siehe Aufgabe 5 des vorherigen Kapitels auf Seite 74) ein!
In Zeile 3 der Prozedur wurde die Menge Zulaessige_Zahlen vereinbart, deren Wertebereich die Zahlen von 0 bis 100 umfaßt. Das Schlüsselwort zur Vereinbarung einer Mengenvariablen lautet **set of**. In allgemeiner Form lautet die Vereinbarung einer Mengenvariablen

var Name: set of Grundmenge

Hinter **set of** wird die **Grundmenge** vermerkt. In der Grundmenge wird beschrieben, aus welchem Datentyp und aus wie vielen Elementen die Menge bestehen soll. Als Datentypen sind BYTE, CHAR, BOOLEAN sowie daraus gebildete Aufzählungs- und Unterbereichstypen zulässig. Die Anzahl der Elemente darf maximal 256 betragen.
Beispiele für Vereinbarung von Mengenvariablen:

```
var  Zeichensatz : set of char;                (1)
     Grossbuchstaben : set of 'A' .. 'Z';      (2)
     MaxZahlenMenge : set of byte;             (3)
     MaximalZahlenMenge : set of 0 .. 255;     (4)
```

Im Fall (1) besteht die Grundmenge aus der Menge der ASCII-Zeichen. Sie umfaßt 256 Elemente. Im Fall (2) wird durch eine Unterbereichsangabe die Menge der Großbuchstaben gebildet. In den Fällen (3) und (4) umfaßt die Grundmenge die ganzen Zahlen von 0 bis 255 einschließlich. Diese weist damit 256 Elemente auf.
Eine Vereinbarung in der Form

```
var   ZahlenMenge : set of 1 .. 256;
```

ist **nicht zulässig**, weil die Zahl 256 nicht im Wertebereich des Datentyps BYTE ist.

3.2 Operationen auf Mengen

3.2.1 Wertzuweisung

Eine der wichtigsten Anwendungen von Mengen besteht in der Prüfung, ob ein Wert in einer Menge enthalten ist. Dazu sind der Mengenvariablen zunächst die zulässigen Elemente der Grundmenge zuzuweisen. In dem obigen Beispiel der Eingabeprozedur stehen in den Variablen UW und OW der kleinste und der größte zulässige Wert. Weist UW den Wert 1, OW den Wert 6 auf, so werden durch die Wertzuweisung

```
5   Zulaessige_Zahl := (.UW..OW.);
```

der Menge Zulaessige_Zahl die Elemente (.1,2,3,4,5,6.) zugewiesen. Die Elemente können in **aufzählender Schreibweise**

```
(.1,2,3,4,5,6.)
```

oder durch **Angabe eines Unterbereichs**

```
(.UW .. OW.)        bzw. [UW..OW]
```

aufgeführt werden. Sie müssen in eckigen Klammern oder in runden Klammern mit Punkt eingeschlossen sein.

3.2.2 Elementprüfung

Mit Hilfe des logischen Operators **IN** wird festgestellt, ob ein Wert in einer Menge enthalten ist. In der Anweisung

15 until **Zahl in Zulaessige_Zahl;**

in Zeile 15 der Prozedur wird geprüft, ob der Wert in der Variablen Zahl in der Menge Zulaessige_Zahl vorkommt.
Das Ergebnis dieser Prüfung ist stets wahr oder falsch. Wenn der Wert, der in der Variablen Zahl steht, in der Menge Zulaessige_Zahl enthalten ist, hat der Bedingungsausdruck

Zahl in Zulaessige_Zahl

den Wahrheitswert wahr (true). Ansonsten weist er den Wahrheitswert falsch (false) auf.

3.2.3 Mengenbildung

Aus Mengen können weitere Mengen gebildet werden. Das folgende Programmbeispiel zur Einteilung von Zahlen verdeutlicht die möglichen Mengenoperationen.

```
 1  program Mengenoperationen;
 2  uses CRT;        (* 4 *)
 3  type Zahlen_bis_10 = set of 1..10;
 4       text          = string (.16.);
 5  const  Zahlen : Zahlen_bis_10 = (.1..10.);
 6  var N                          : integer;
 7      Primzahlen, UngeradeZahlen,
 8      GeradeZahlen, GeradePrimzahl : Zahlen_bis_10;
 9  procedure Ausgabe (Atext : text;
                       Menge : Zahlen_bis_10);
10  var Z : integer;
11  begin
12    write (Atext);
13    for Z := 1 to 10 do
14      if Z in Menge
          then write(Z:3);
15    writeln;
16  end;
```

```
17  begin                                  (* HAUPTPROGRAMM *)
18    ClrScr;
19    GeradeZahlen := (. .);        (* leere Menge *)
20    UngeradeZahlen := (.  .);
21    for N := 1 to 10 do
22    begin
23      if odd(N)
24        then UngeradeZahlen := UngeradeZahlen + (.N.)
25        else GeradeZahlen := GeradeZahlen + (.N.);
26    end;
27    Ausgabe ('Gerade Zahlen   ',GeradeZahlen);
28    Ausgabe ('Ungerade Zahlen',UngeradeZahlen);
29    PrimZahlen := UngeradeZahlen - (.1,9.) + (.2.);
30    Ausgabe ('Primzahlen      ',Primzahlen);
31    GeradePrimzahl:= PrimZahlen * GeradeZahlen;
32    Ausgabe ('Gerade Primzahl ',GeradePrimzahl);
33  end.
```

Wenn Sie das Programm eingeben und starten, erscheint folgende Ausgabe auf dem Bildschirm:

```
Gerade Zahlen     2  4  6  8 10
Ungerade Zahlen   1  3  5  7  9
Primzahlen        2  3  5  7
Gerade Primzahl   2
```

Erläuterungen

Wie kommen diese Ausgaben zustande? In Zeile 3 wird eine typisierte Konstante mit dem Namen Zahlen vereinbart, deren Datentyp Zahlen_bis_10 ist und der die Menge der Zahlen von 1 bis 10 zugewiesen wird. Die Menge Zahlen besteht also aus folgenden Elementen:

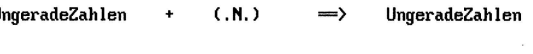
Zahlen	1	2	3	4	5	6	7	8	9	10

Im Programm soll die Menge der geraden und ungeraden Zahlen sowie die Menge der Primzahlen und der geraden Primzahlen gebildet werden. Entsprechende Mengenvariablen werden in Zeile 7 und 8 mit dem Datentyp Zahlen_bis_10 deklariert.
Bevor in der FOR-Anweisung die Menge der geraden und ungeraden Zahlen gebildet wird, wird in den Zeilen 19 und 20 den beiden Mengen jeweils die leere Menge (.''.) zugewiesen. Dadurch werden sie zu leeren Mengen, die kein Element haben. Diese Operation ist notwendig, weil in der folgenden FOR-Anweisung die Menge der geraden bzw. ungeraden Zahlen durch "Summierung" gebildet wird. Genauer ausgedrückt: Die Wertzuweisung.

UngeradeZahlen := UngeradeZahlen + (.N.)

bewirkt, daß die Menge der ungeraden Zahlen und die Menge (.N.) in der Menge UngeradeZahlen vereinigt werden.

Beispiel:



Mit dem Operationszeichen + wird die **Vereinigungsmenge** erzeugt.
Die Menge der geraden Zahlen wird in gleicher Weise gebildet.
Um festzustellen, welche Zahl gerade bzw. ungerade ist, wird in Zeile 23 die Funktion **ODD** angewendet. Odd(N) liefert den Wert true, wenn der Inhalt von N eine ungerade Zahl ist. Bei einer geraden Zahl weist Odd den Wert false auf.
Die Menge der Primzahlen ergibt sich aus der Menge der ungeraden Zahlen ohne die Menge (.1,9.), vereinigt mit der Menge (.2.).

```
29  PrimZahlen := UngeradeZahlen - (.1,9.) + (.2.);
```

Die **Differenzmenge** wird durch den Mengenoperator – erzeugt.
Um die Menge der geraden Primzahlen zu erhalten, ist die Schnittmenge der Primzahlen-Menge mit der Menge der geraden Zahlen zu bilden. Die Schnittmenge enthält die Elemente, die sowohl in der

Menge der Primzahlen als auch in der Menge der geraden Zahlen enthalten sind.

```
31  GeradePrimzahl:= PrimZahlen * GeradeZahlen;
```

Die **Schnittmenge** wird durch den Mengenoperator * gebildet.
Die so gebildeten Mengen werden mittels der Prozedur Ausgabe auf dem Bildschirm angezeigt. Die auszugebende Menge wird in dem Programmbeispiel an die Prozedur Ausgabe übergeben.
Eine Menge kann nicht direkt ausgegeben werden. Deshalb muß man eine Hilfskonstruktion über die FOR-Anweisung anwenden. Dabei nimmt die Laufvariable N alle Werte an, die als Elemente der Grundmenge vorkommen können.

```
13  for Z := 1 to 10 do
14  if Z in Menge then write(Z:3);
```

Mittels einer IN-Abfrage wird geprüft, ob der aktuelle Inhalt von Z in der betreffenden Menge enthalten ist. Der Wert von Z, der als Element in der Menge enthalten ist, wird ausgegeben.

3.3 Zusammenfassung

- Eine **Menge** besteht aus einer bestimmten Anzahl von Elementen eines **abzählbaren Datentyps**.
- Unter Verwendung der Datenstruktur Menge (set) können Programme auf relativ einfache Weise gegen fehlerhafte und falsche Eingaben abgesichert werden.
- Eine Menge kann entweder direkt oder über eine Mengenvariable in ein Programm eingebracht werden.
  Beispiel für eine direkte Einbeziehung:
  if Wert in (. 1..5 .)
- Bei der Einbringung von Mengen mittels Mengenvariablen empfiehlt sich folgende Vorgehensweise:

1. Vereinbarung eines Mengentyps:	type Mengentyp = set of Grundmenge
2. Vereinbarung der Mengenvariablen:	var MengenName: Mengentyp

3. Zuweisung der Menge an die Mengenvariable

Menge := (.El_1 .. El_n.)
oder
Menge := (.El_1,El_2,El_3.)

- Die Grundmenge kann durch Angabe des ersten und letzten Elements charakterisiert werden, wenn die Elemente in einer Reihenfolge stehen: 'a' .. 'z'. Besteht eine solche Reihenfolge nicht, müssen die Elemente einzeln aufgelistet werden: ('a','e','i','o','u').
- In Turbo-Pascal darf eine Menge aus höchstens **256** Elementen bestehen. Hinweis: Die leere Menge ist Element jeder Menge!
- Die **leere Menge** (. .) besitzt kein Element.
- Durch **Vereinigung, Durchschnitt und Differenzbildung** können aus Mengen weitere Mengen gebildet werden. Die Operationszeichen sind:

		A := (.1,2.); B := (.2,3,4.)	
Vereinigungsmenge:	+	C := A+B	C enthält (.1,2,3,4.)
Durchschnittsmenge:	*	D := B*A	D enthält (.2.)
Differenzmenge:	–	E := A–D	E enthält (.1.)

Logische Operationen auf Mengen:

	Prüfung auf	Beispiel	Ergebnis
=	Mengengleichheit	A = B	falsch
<=	Teilmenge	D <= A	richtig
>=	Obermenge	C >= B	richtig
<>	Mengenungleichheit	E <> B	richtig
IN	Enthaltensein	3 IN C	richtig

3.4 Übungen

1. Aufgabe
Als Eingabewerte sind nur die Ziffern 1,3,5,6 zulässig. Vervollständigen Sie die Eingabe-Schleife!

```
repeat
  read (Wert);
until...
```

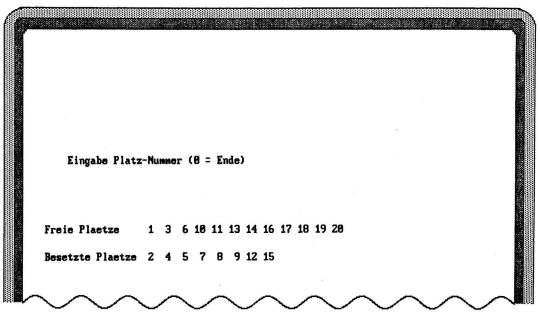
2. Aufgabe
Die Menge GueltigeZeichen soll in verschiedenen Programmsituationen die Elemente ('a','f','g','k'.) oder ('b','e','m'.) aufweisen. Wie muß die Menge GueltigeZeichen vereinbart werden?

3. Aufgabe
Vervollständigen Sie die FOR-Schleife, über die ausgegeben wird, welche Elemente in der Menge GueltigeZeichen (s. Aufgabe 2) enthalten sind.

```
for Zeichen := ___ to ___ do
    if _____ in _____________
    then _____________________ ;
```

4. Aufgabe
Um den Überblick zu behalten, welche der 20 Plätze in einem Reisebus bereits reserviert bzw. noch frei sind, will ein Reiseunternehmen ein Programm einsetzen. Wie aus der nachstehenden Bildschirmabbildung zu ersehen ist, zeigt das Programm an:

Platznummer der freien Plätze
Platznummer der belegten Plätze



Nach jeder Platzbuchung wird der aktuelle Stand angezeigt. Wurde die Nummer eines Platzes eingegeben, der bereits reserviert ist, soll das Programm einen Fehler melden.

Die Platzbuchungen können so lange vorgenommen werden, bis entweder als Platznummer eine Null eingegeben wurde oder alle Plätze gebucht sind. Zum Abschluß des Programms wird die Anzahl der reservierten Plätze angezeigt.
Folgender Grob-Algorithmus wird vorgegeben:

Algorithmus Reisebus	
Freie Plätze := Plätze	
Besetzte Plätze := (. .) (*leere Menge*)	
Ausgabe Freie Plätze	
Eingabe Platznummer	
Platznummer in Freie Plätze	
ja	nein
Platznummer aus Freie Plätze austragen	Ausgabe Fehlermeldung
Platznummer in Besetzte Plätze eintragen	
Ausgabe Freie Plätze	
Ausgabe Besetzte Plätze	
noch freie Plätze vorhanden	
ja	nein
Eingabe Platznummer	./.
bis Platznummer = 0 oder kein freier Platz vorhanden	
Ermitteln der Anzahl der besetzten Plätze	
Ende	

5. Aufgabe
Entwickeln Sie ein Programm, das einen Text, der mit maximal 40 Zeichen Länge eingegeben wurde, ohne Vokale ausgibt!

3.5 Lösungen

1. Aufgabe

```
repeat
  read (Wert);
until Wert in (.1,3,5,6.);
```

2. Aufgabe

1. Vereinbarung Mengentyp: type Zeichensatz = set of 'a'..'z';
2. Vereinbarung der Mengenvariablen: var GueltigeZeichen: Zeichensatz;
3. Zuweisung der Mengen im Anweisungsteil des Programms: GueltigeZeichen := (. 'a','f','g','k'.) bzw. GueltigeZeichen := (. 'b','e','m'.)

3. Aufgabe

```
for Zeichen := 'a' to 'z' do
  if Zeichen in GueltigeZeichen
    then write (Zeichen);
```

4. Aufgabe

(siehe gegenüberliegende Seite)

```
program Reisebus;
uses CRT;              (* 4 *)
type  PlatzNr  = set of 1..20;
      string17 = string (.17.);
const Plaetze       : PlatzNr = (.1..20.);
var   Besetzt, Frei : PlatzNr;
      Nr, M, Summe  : integer;


procedure Ausgabe (Sp,Zl : integer;
                   Atext : string17;
                   Menge : PlatzNr);
var N : integer;
begin
  Gotoxy (Sp,Zl); ClrEol;
  write (Atext);
  for N := 1 to 20 do
    if N in Menge
      then write (N:3);
  writeln;
end;


begin                                   (* HAUPTPROGRAMM *)
  ClrScr;
  Frei := Plaetze;
  Ausgabe (1,15,'Freie Plaetze   ', Frei);
  Besetzt := (. .);
  Gotoxy (5,10); write ('Eingabe Platz-Nummer (0 = Ende) ');
  ClrEol; read (Nr);
  repeat
    if Nr in Frei
      then begin
        Frei := Frei - (.Nr.);
        Besetzt := Besetzt + (.Nr.);
        Ausgabe (1,15,'Freie Plaetze   ', Frei);
        Ausgabe (1,17,'Besetzte Plaetze', Besetzt);
      end
      else begin
        Gotoxy (42,10); write (' besetzt'); delay(400)
      end;
    if not (Frei = (. .))
      then begin
        Gotoxy (5,10); write ('Eingabe Platz-Nummer (0 = Ende) ');
        ClrEol; read (Nr);
      end;
  until (Nr =0) or (Frei = (. .));
  Summe := 0;
  for M := 1 to 20 do
    if M in Besetzt
      then Summe := Summe + 1;
  Gotoxy (5,22); write ('Anzahl reservierter Plaetze:',Summe:4);
end.
```

5. Aufgabe

```
1  program Vokal_Unterdruecker;
2  uses CRT;          (* 4 *)
3  type Zeichensatz = set of 'A'..'z';
4  var  Text        : string (.40.);
5       Buchstabe   : Zeichensatz;
6       N           : integer;

7  begin
8    ClrScr;
9    Buchstabe := (.'a','e','i','o','u','A','E','I','O','U'.);
10   write ('Eingabe Text: '); readln (Text);
11   for N := 1 to length (Text) do
12     if not (Text (.N.) in Buchstabe)
13       then write (Text(.N.));
14 end.
```

4 DATENSTRUKTUR VERBUND (RECORD)

Einführungsbeispiel: Die Lotto-Tippgemeinschaft sitzt am Montagmorgen in der Kantine zusammen. Paul seufzt: "Mit unserem gemeinsamen Lottoschein haben wir schon wieder nichts gewonnen. Dabei hatte ich das sichere Gefühl, daß meine Tippreihe uns das große Glück bringt." Seine 4 Kollegen stimmen in das Klagelied ein. "Mensch", sagt Paul, "ich habe eine Idee!" Er wendet sich an seinen Kollegen Michael: "Michael, in eurer Abteilung steht doch ein PC. Da kannst du doch den Computer ausrechnen lassen, wessen Tippreihe als erste 6 Richtige haben wird." "Nun, Glück läßt sich nicht berechnen, aber wenn es euch Spaß bereitet", antwortet Michael, "so schreibe mir jeder seine 6 Zahlen und seinen Namen auf einen Zettel!"
Zur Lösung dieses Problems wird die Datenstruktur **RECORD** benötigt. RECORD bedeutet in der Fachsprache "Satz von Angaben". Gleichbedeutend hiermit ist der Begriff **Verbund**.
Da diese Datenstruktur im Rahmen der Dateiverarbeitung von sehr großer Bedeutung ist, wird schrittweise dargelegt, wie ein Verbund aufgebaut und behandelt wird.

4.1 Charakteristik der Datenstruktur Verbund

Für die Lottoauswertung müssen pro Spieler Name und Tippreihe eingegeben werden. Zur Speicherung des Namens wird eine 20stellige Zeichenkette vorgesehen. Für die Aufnahme der Tippreihe, die aus 6 Zahlen im Bereich von 1 bis 49 besteht, ist ein Feld mit 6 Elementen vom Typ BYTE einzurichten.

```
Name     : string (.20.)
Tippreihe : array (.1..6.) of byte
```

Diese beiden Angaben gehören in Hinblick auf das Lottoproblem logisch zusammen. Sie bilden eine logische Einheit. Eine solche Zusammenfassung von Daten unterschiedlichen Typs wird als **Datensatz** bezeichnet.
Zur Darstellung eines Datensatzes kennt Pascal die Datenstruktur **RECORD**. Im Gegensatz zur Datenstruktur **ARRAY** dürfen die Bestandteile eines RECORDS **unterschiedliche Datentypen** aufweisen.

Ein RECORD bzw. Verbund ist die Zusammenfassung von Daten unterschiedlichen Typs zu einer neuen Einheit.

4.2 Vereinbarung und Zugriff

Eine Variable, die aus Elementen unterschiedlichen Typs besteht, nennt man Verbundvariable oder Recordvariable. Wie eine Verbundvariable vereinbart wird, ist aus dem nachstehenden Programm zu ersehen, mit dem der Name eines Spielers und seine Tippreihe eingegeben und am Bildschirm angezeigt wird.

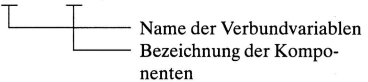
```
1  program Lotto1;
2  uses Crt;        (* 4 *)
3  var Spieler : record
                   Name      : string (.20.);
                   Tippreihe : array (.1..6.) of byte;
                 end;
4     N        : integer;

5  begin
6    ClrScr;
7    Gotoxy (5,10); write ('Name: '); readln (Spieler.Name);
8    for N := 1 to 6 do
9    begin
10     Gotoxy (5,12+N); write (N:2,'. Zahl: ');
11     readln (Spieler.Tippreihe(.N.));
12   end;
13   writeln;
14   write (Spieler.Name);
15   for N := 1 to 6 do write (Spieler.Tippreihe(.N.):6);
16   writeln;
17 end.
```

In Zeile 3 wurde die Verbundvariable Spieler vereinbart. Das Schlüsselwort dazu lautet **record**. Dahinter werden die Bestandteile (Komponenten) angegeben, aus denen sich der Verbund zusammensetzt. Zu jedem Bestandteil wird der Datentyp angegeben. Ein Semikolon trennt die Vereinbarung einer Komponenten von der nächsten. Ein **end;** schließt die Vereinbarung der Verbundvariablen ab. In allgemeiner Form lautet die Vereinbarung einer RECORD-Variablen:

```
var   Variablenname: record Komponente_1: Datentyp;
                            Komponente_2: Datentyp;
                            ...
                            Komponente_n: Datentyp;
                     end;
```

Um auf die Komponente einer Verbundvariablen zugreifen zu können, müssen der Name der Verbundvariablen und der Komponentenname angegeben werden. Die beiden Adreßbestandteile sind durch einen **Punkt** zu trennen.

Beispiel: readln (Spieler.Name);

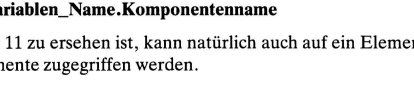


Über die Bezeichnung Spieler wird die Verbundvariable angesprochen. Über die Bezeichnung Name wird der betreffende Bestandteil adressiert. Um also auf eine Komponente einer Verbundvariablen zugreifen zu können, sind anzugeben:

Verbundvariablen_Name.Komponentenname

Wie aus Zeile 11 zu ersehen ist, kann natürlich auch auf ein Element einer Komponente zugegriffen werden.

```
11  readln (Spieler.Tippreihe(.N.));
```



WITH-Anweisung

Um auf eine Komponente eines RECORDS zugreifen zu können, muß stets der Name des RECORDS und der der Komponente angegeben werden. Das ist mit viel Schreibarbeit verbunden und somit lästig. Deshalb gibt es in Pascal die Möglichkeit, die Schreibweise mittels einer **WITH**-Anweisung abzukürzen. Das wird am Beispiel der Programmzeilen 14 bis 15 dargestellt. In den Schreib-Anweisungen wird stets auf die Verbundvariable Spieler Bezug genommen. Diesen Teil der Adreßangabe kann man mit der WITH-Anweisung "vorziehen". Die Programmzeilen lauten dann:

```
with Spieler do
begin
  write (Name);
  for N := 1 to 6 do write (Tippreihe (.N.) :6);
end; (* of with *)
```

Hinter with wurde die Bezeichnung der Verbundvariablen Spieler vermerkt. Damit wird dem Rechner mitgeteilt, daß bei jeder Variablen in den nachfolgenden Anweisungen zu prüfen ist, ob sie Bestandteil der Verbundvariablen Spieler ist. Falls sie Komponente der Verbundvariablen ist, soll das System die Operation auf die Komponente der Verbundvariablen beziehen. Ist in den Anweisungen eine Variable aufgeführt, die nicht zu einem Verbund gehört, so wird sie als "normale" Variable behandelt.

Aufgrund der automatischen Zugehörigkeitsprüfung nach einer WITH-Anweisung ist es wichtig, daß "normale" Variablen und Bestandteile von Verbundvariablen nicht dieselben Bezeichnungen haben. Ansonsten entstehen häufig Situationen, in denen man der Namensgleichheit wegen die WITH-Anweisung nicht anwenden kann.

In allgemeiner Form lautet die WITH-Anweisung:

with Verbundvariable **do** Anweisung;

Der **Gültigkeitsbereich** der WITH-Anweisung erstreckt sich nur auf die Anweisung, die dem **do** folgt. Soll die WITH-Anweisung für mehrere aufeinanderfolgende Anweisungen gelten, so muß der Gültigkeitsbereich durch begin und end zu einem Block zusammengefaßt werden.

```
with Verbundvariable do
begin
  Anweisung_1;
  Anweisung_2;
  ...
end;  (* of with *)
```

4.3 Zusammengesetzte Datenstrukturen: ARRAY of RECORD

Mit dem obigen Programm kann nur ein Datensatz eingegeben werden. Das reicht aber nicht, da die Tippgemeinschaft aus insgesamt 5 Spielern besteht. Deshalb ist eine Datenstruktur zu schaffen, die entsprechend viele Einträge zuläßt und programmtechnisch gut zu handhaben ist.
Da die Datensätze in ihrem Aufbau für alle 5 Spieler gleich sind, können sie in einem eindimensionalen Feld zusammengefaßt werden. Wenn das eindimensionale Feld mit dem Namen TipTeam bezeichnet wird, lautet die Vereinbarung:

```
var TipTeam: array (.1..5.) of record
                                Name : string (.20.);
                                Tippreihe: array (.1..6.) of byte;
                                end;
```

Durch diese Variablenvereinbarung wird ein eindimensionales Feld gebildet, dessen Elemente Verbundvariablen sind. Es liegt hier ein **ARRAY OF RECORD** vor.
Die Variablenvereinbarung von TipTeam ist relativ komplex. Zur Vereinfachung nutzen wir die Möglichkeit der Typenvereinbarung. Zunächst wird für den Datensatz eines Spielers ein eigener Datentyp vereinbart.

```
type SpielerSatz = record Name: string(.20.);
                          Tippreihe: array (.1..6.) of byte;
                          end;
```

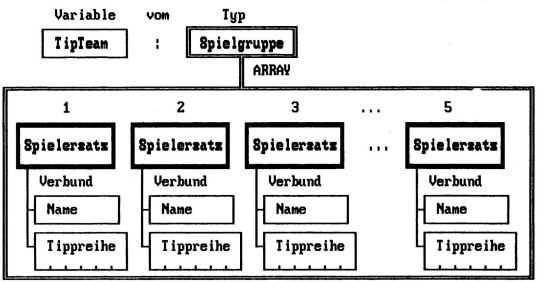
Unter Anwendung des Verbundtyps SpielerSatz wird der Feldtyp SpielGruppe geschaffen:

```
type SpielGruppe = array (. 1..5 .) of SpielerSatz;
```

Diese Datentypvereinbarung wird nun dazu benutzt, die Variable TipTeam zu definieren:

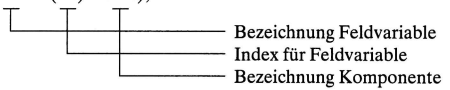
var TipTeam : SpielGruppe;

Die Datenorganisation kann man sich dann wie folgt vorstellen:



Die obige Abbildung verdeutlicht, daß der Datentyp der Variablen TipTeam ein eindimensionales Feld ist, dessen Elemente jeweils aus einem Verbund von Name und Tippreihe bestehen.
Um den Namen des ersten Spielers eingeben zu können, müßte der Lesebefehl lauten:

readln (TipTeam (.1.).Name);



Die Variable TipTeam ist ein eindimensionales Feld. Deshalb muß bei TipTeam der Index angegeben werden, über den das einzelne Element des Feldes TipTeam adressiert wird. In allgemeiner Form wird der Zugriff auf eine solche Datenstruktur ARRAY of RECORD wie folgt formuliert:

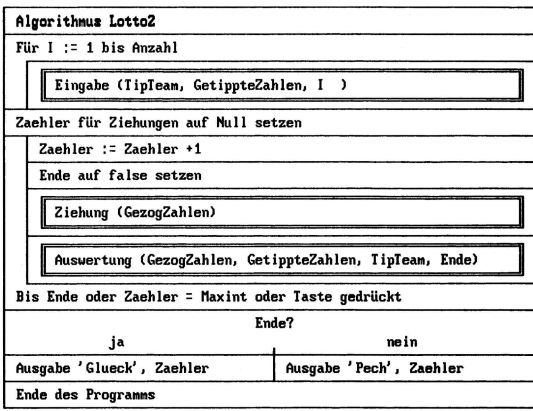
Feldvariablen_Name (.Index.).Komponentenbezeichnung

Nach der Festlegung der Datenstruktur kann nun der Algorithmus zur Lösung der Problemstellung entwickelt werden. Das Programm soll ermitteln, nach wie vielen Ausspielungen eine Tippreihe voraussichtlich "6 Richtige" haben wird.

Algorithmus

Der Ablauf des Programms läßt sich wie folgt skizzieren:

1. Es werden die Datensätze der Spieler eingegeben. Dabei wird für jeden Spieler aus den Zahlen der Tippreihe die Menge GetippteZahlen gebildet.
2. Per Zufallszahlengenerator werden die 6 Gewinnzahlen ermittelt, aus denen die Menge der gezogenen Zahlen gebildet wird.
3. Die Menge der gezogenen Zahlen wird mit der Menge der getippten Zahlen der einzelnen Spieler verglichen. Wird eine Mengengleichheit festgestellt, ist die Aufgabe gelöst. Bei Mengenungleichheit muß die Verarbeitung mit dem zweiten Schritt wiederholt werden, es sei denn, daß die maximale Anzahl von Durchläufen erreicht ist oder der Programmlauf durch einen Tastendruck beendet wird.



```
Prozedur Eingabe (var TippGem; var TippZahlen; X)
Menge TippZahlen := (. .)        (*Leere Menge *)
Eingabe Name des Spielers X
Für N := 1 bis Anzahl
    Eingabe TippZahlen (.N.) des Spielers X
    TippZahl(.N.) in Menge TippZahlen des Spielers X eintragen
Ende der Prozedur
```

```
Prozedur Ziehung (var GezogeneZahlen)
Menge GezogeneZahlen := (. .)        (*Leere Menge *)
Mächtigkeit := 0
    Bilde Zufallszahl
        Zufallszahl in der Menge GezogeneZahlen?
        ja                                          nein
    Mächtigkeit um 1 erhöhen                        ./.
    Zufallszahl in Menge GezogeneZahlen eintragen
Bis Mächtigkeit der Menge GezogeneZahlen = 6
Ende der Prozedur
```

```
Prozedur Auswertung (var GezogeneZahlen; var TippZahlen;
                     var TippGem; var Fertig)
N := 0
    N := N + 1
        GezogeneZahlen = TippZahlen (.N.)?
        ja                                          nein
    Ausgabe TippGem (.N.).Name                      ./.
    Fertig := true
Bis N = Anzahl
Ende der Prozedur
```

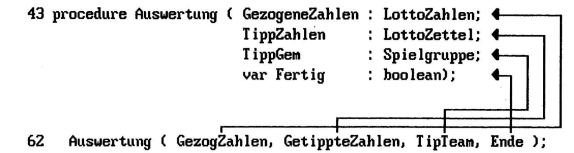
Programm

```
1  program Lotto2;
2  uses Crt;       (* 4 *)
3  const Anzahl = 5;
4  type  Lottozahlen = set of 1..49;
         SpielerSatz = record
                         Name      : string (.20.);
                         Tippreihe : array (.1..6.) of byte;
                       end;
5        Spielgruppe = array (.1..Anzahl.) of SpielerSatz;
6        LottoZettel = array (.1..Anzahl.) of LottoZahlen;

7  var   TipTeam        : Spielgruppe;
8        I, Zaehler     : longint;
9        GetippteZahlen : LottoZettel;
10       GezogZahlen    : Lottozahlen;
12       Ende           : boolean;


13 procedure Eingabe (var TippGem     : Spielgruppe;
                      var  Tippzahlen : Lottozettel;
                           N          : integer);
14 var N : integer;
15 begin
16   Tippzahlen (.N.) := (. .);        (* Zuweisung leere Menge *)
17   with TippGem (.N.) do
18   begin
19     ClrScr;
20     Gotoxy (5,10); write ('Name: '); readln (Name);
21     for N := 1 to 6 do
22     begin
23       Gotoxy (5,12+N); write (N:2,'. Zahl: ');
24       readln (Tippreihe(.N.));
25       Tippzahlen (.N.) := Tippzahlen (.N.) + (.Tippreihe (.N.).);
26     end;
27   end;                                (* of with *)
28 end;


29 procedure Ziehung (var GezogeneZahlen: LottoZahlen);
30 var Zufallszahl, Maechtigkeit : integer;
31 begin
32   GezogeneZahlen := (.  .); Maechtigkeit := 0;
33   repeat
34     randomize;
35     Zufallszahl := random (49) +1;
36     if not ( Zufallszahl in GezogeneZahlen)
37       then begin
38         Maechtigkeit := Maechtigkeit + 1;
39         GezogeneZahlen := GezogeneZahlen + (. Zufallszahl.);
40       end;
```

```
41    until Maechtigkeit = 6;
42  end;


43  procedure Auswertung ( GezogeneZahlen : LottoZahlen;
                           TippZahlen     : LottoZettel;
                           TippGem        : Spielgruppe;
                           var Fertig     : boolean);
44  var N : integer;
45  begin
46    N := 0;
47    repeat
48      N:=  N+1;
49      if GezogeneZahlen = TippZahlen (.N.)
50        then begin
51          writeln ('6 Richtige hat ',TippGem(.N.).Name);
52          Fertig := true
53        end;
54    until N = Anzahl;
55  end;


56  begin                                  (* HAUPTPROGRAMM *)
57    for I := 1 to Anzahl do  Eingabe(TipTeam,GetippteZahlen,I);
58    Zaehler := 0;
59    repeat
60      Ende := false;  Zaehler := Zaehler +1;
61      Ziehung( GezogZahlen);
62      Auswertung(GezogZahlen, GetippteZahlen, TipTeam, Ende);
63      writeln (Zaehler);
64    until Ende or (Zaehler = maxint) or keypressed;
65    if Ende
        then writeln ('Viel Glueck nach ',Zaehler,' Versuchen!')
66      else writeln (Zaehler,'-mal versucht und kein Glueck!');
67    Delay (2000);
68  end.
```

Erläuterungen

In den Ausführungen zur Datenstruktur Feld wurde dargelegt, daß ein Feld nicht als Ganzes in einer einfachen WRITE-Anweisung auf einen Drucker oder Bildschirm ausgegeben werden kann. Dazu bedarf es vielmehr einer Schleifenkonstruktion. Anders verhält es sich, wenn eine Variable, die vom Typ Menge, Feld oder RECORD ist, in eine andere Variable kopiert wird. Hier kann eine Variable als Ganzes übertragen werden. Das ist beispielsweise aus dem Aufruf der Prozedur Auswertung zu ersehen:



Voraussetzung für eine solche "Ausgabe" von Variableninhalten an eine andere Variable ist, daß beide Variablen genau denselben Datentyp haben.

> Eine Variable vom Typ RECORD oder ARRAY kann nur dann insgesamt auf eine andere Variable übertragen bzw. zugewiesen werden, wenn die beiden Variablen genau dieselbe Datenstruktur aufweisen.

Da in dem Programm sehr oft Variablen an die Prozeduren übergeben werden, mußten einige Typenvereinbarungen getroffen werden. Mit der Typenvereinbarung

4 type Lottzahlen = set of 1..49;

wird die Menge der Lottozahlen mit der Grundmenge der Zahlen von 1 bis 49 festgelegt.
Der Feldtyp Lottozettel

6 LottoZettel = array (.1..Anzahl.) of LottoZahlen;

beschreibt den Aufbau des eindimensionalen Feldes, in dem die Menge der von den Spielern getippten Zahlen gebildet wird.
In der Prozedur Eingabe ist zu erkennen, daß sich die Schreibweise durch die WITH-Anweisung

17 with TippGem (.X.) do

wesentlich vereinfacht. Der Gültigkeitsbereich der WITH-Anweisung erstreckt sich von Zeile 17 bis Zeile 27.
Mit der Anweisung

25 Tippzahlen (.X.) := Tippzahlen (.X.) + (.Tippreihe (.N.).);

wird durch Mengenvereinigung die Menge der vom Spieler getippten

Zahlen gebildet. Damit die Mengenvereinigung ordnungsgemäß erfolgt, ist die Menge, in der die Vereinigung erfolgt, vorher zu "löschen". Das geschieht durch Zuweisung der leeren Menge:

```
16  Tippzahlen (.X.) := (. .);
```

Entsprechend wird auch bei der Bildung der Menge der gezogenen Zahlen verfahren.
Werden in kurz aufeinanderfolgenden RANDOM-Anweisungen Zufallszahlen gebildet, so kommt es häufig zur Ziehung derselben Zahlen, auch wenn jeweils vorher der Zufallszahlengenerator durch die Anweisung RANDOMIZE initialisiert wurde. RANDOMIZE bildet nämlich den Startwert für die folgende RANDOM-Anweisung aus der Systemzeit, womit der Zufall in Wirklichkeit keiner ist. Ohne RANDOMIZE würde in der Anweisung

```
35  Zufallszahl := random (49) +1;
```

stets dieselbe Folge von "Zufallszahlen" erzeugt. Da die RANDOM-Funktion mit dem Integer-Parameter (49) die Zahl Null als kleinste und die Zahl 48 als größte Zufallszahl bildet, muß die Zufallszahl um 1 erhöht werden.
In der REPEAT-Schleife (Zeile 33–41) wird die Menge der gezogenen Zahlen gebildet. Die Schleife wird erst verlassen, wenn in der Menge 6 Elemente eingetragen wurden und sie somit die Mächtigkeit 6 besitzt. Zur Mengenbildung darf nicht einfach eine Schleife verwendet werden, die 6mal durchlaufen wird. Es ist nämlich nicht gewährleistet, daß in einer 6mal durchlaufenen Schleife tatsächlich 6 verschiedene Zahlen erzeugt werden.
Durch die Anweisung

```
36  if not (Zufallszahl in GezogeneZahlen)
```

wird sichergestellt, daß nur dann die Variable Mächtigkeit erhöht und eine Mengenvereinigung vorgenommen wird, wenn die Zufallszahl nicht schon in der Menge der gezogenen Zahlen enthalten ist.
Die Schleifenaustrittsbedingung der REPEAT-Anweisung im Hauptprogramm

```
64  until Ende or (Zaehler = maxint) or keypressed;
```

weist drei unterschiedliche Austrittskriterien auf. So kann die Schleife verlassen werden, wenn Zaehler den Wert **maxint** (= maxi-

male Integerzahl) erreicht hat. **maxint** ist eine Systemkonstante mit dem Wert 32767. Die wichtigste Austrittsbedingung ist hier **keypressed**. Dadurch können Sie, wenn der Rechner auch nach längerer Laufzeit noch keinen Lottokönig ermittelt hat, den Programmlauf abbrechen.
Die Funktion **keypressed** fragt den Tastaturpuffer ab, ob eine Taste, die ein lesbares Zeichen erzeugt, gedrückt worden ist. Die Funktion keypressed weist den Wert true auf, wenn eine "Zeichen-Taste" betätigt wurde. Das Drücken einer Funktionstaste beispielsweise hat also keine Wirkung.
Das Programm Lotto2 ist ein gutes Beispiel dafür, wie in Pascal aus den Datenstrukturen Feld, Menge und Verbund weitere Datenstrukturen aufgebaut werden. So stellt der Datentyp LottoZettel einen ARRAY of SET dar. Der Datentyp Spielgruppe ist noch komplexer aufgebaut. Er ist ein ARRAY, dessen Elemente vom Typ RECORD sind. Innerhalb des RECORDS tritt ein ARRAY als Komponente auf. In Pascal ist es also möglich, die programminterne Datenstruktur exakt der Struktur der Daten, wie sie sich aus dem gestellten Problem ergibt, anzupassen.

4.4 Zusammenfassung

- Die Datenstruktur **Verbund (Record)** gestattet es, Variablen (Komponenten) unterschiedlichen Datentyps zu einer neuen Einheit zusammenzufassen.
- Die Vereinbarung einer Verbundvariablen wird mit Hilfe des Schlüsselwortes **record** vorgenommen. Die Record-Vereinbarung wird durch ein **end;** abgeschlossen.

```
var Verbundvariable: record Komponente_1: Datentyp;
                            Komponente_2: Datentyp;
                            ...
                     end;
```

- Der Zugriff auf eine Komponente einer Verbundvariablen wird über eine zusammengesetzte Adreßangabe vorgenommen:

Name_Verbundvariable.Komponentenbezeichnung

- Bei Verwendung der **WITH**-Anweisung kann die Adressierung einer Komponenten vereinfacht geschrieben werden.

- Die Datenstruktur Verbund kann zum Aufbau weiterer Datenstrukturen verwendet werden.
- Sind Verbundvariablen Elemente eines eindimensionalen Feldes, so lautet die Adreßangabe für eine Komponente des Verbundes:

Verbundvariablen_Name (.Feldindex .).Komponentenname

- Die Komponenten der Datenstruktur Verbund können wiederum strukturierte Daten sein.
- Eine strukturierte Variable kann einer anderen strukturierten Variablen als Einheit zugewiesen werden, wenn beide Variablen **genau denselben Datentyp** haben.
- Die Funktion **KEYPRESSED** prüft den Tastaturpuffer, ob eine Taste, die ein lesbares Zeichen erzeugt, gedrückt wurde. Sie gibt den Wert true aus, wenn eine Taste betätigt wurde.
- Mittels **RANDOMIZE** wird der Zufallszahlengenerator initialisiert.
- Die Funktion **RANDOM (Integer-Wert)** bildet eine Zufallszahl vom Typ INTEGER, für die gilt:

  0 <= Zufallszahl < Integer-Wert

  Wird bei der **RANDOM-Funktion kein Integer-Parameter** angegeben, wird eine Zufallszahl vom Typ REAL erzeugt, für die gilt:

  0 <= Zufallszahl < 1

4.5 Übungen

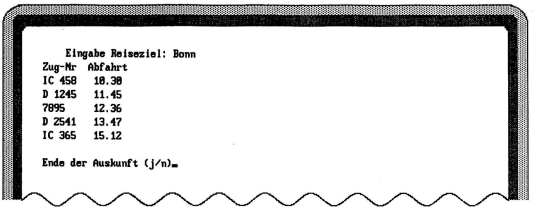
1. Aufgabe
Worin unterscheiden sich die Datenstrukturen Feld und Verbund?

2. Aufgabe
Welche zwei Adreßbestandteile müssen angegeben werden, um auf eine Komponente eines Verbundes zugreifen zu können?

3. Aufgabe
Vereinbaren Sie eine Verbundvariable für einen Artikelsatz, der aus Artikelnummer, Artikelbezeichnung, Lagermenge und Einzelpreis besteht!

4. Aufgabe
Vereinbaren Sie ein eindimensionales Feld, dessen Elemente aus 50 Artikelsätzen bestehen! Zum Aufbau des Artikelsatzes siehe Aufgabe 3. Verwenden Sie die Möglichkeit der Typvereinbarung!

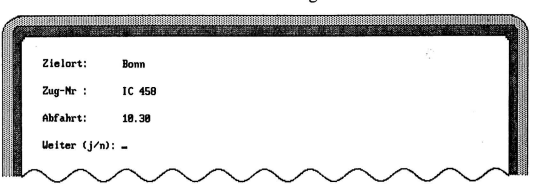
5. Aufgabe
Entwickeln Sie ein Programm, mit dem ein Reisebüro in einer Großstadt Auskunft über bestehende unmittelbare Zugverbindungen zu einer anderen Großstadt geben kann! Der Arbeitsablauf ist so zu gestalten, daß der Sachbearbeiter den gewünschten Zielort über die Tastatur eingibt. Auf dem Bildschirm sollen daraufhin die Zugverbindungen mit Zug-Nummer und Abfahrtszeit angezeigt werden.



Damit die Zugdaten angezeigt werden können, müssen sie natürlich erst eingegeben werden. Das Programm muß also eine Prozedur enthalten, mit der bis zu 30 Zug-Datensätze erfaßt werden können. Ein Zug-Datensatz besteht aus

Zielort: 25stellige Zeichenkette
Zug-Nr.: 5stellige Zeichenkette
Abfahrtszeit: real

Bildschirmmaske für die Datenerfassung:



6. Aufgabe
Entwickeln Sie die Datenstruktur für die Leistungsdaten einer Schulklasse! Die Schulklasse bestehe aus 25 Schüler(innen), die in 9 Fächern unterrichtet werden. In diesen Fächern werden bis zu 6 Klassenarbeiten geschrieben. Für den Schülernamen und die Fachbezeichnung sind jeweils Zeichenketten mit einer Länge von 20 Stellen vorzusehen.

4.6 Lösungen

1. Aufgabe
In der Datenstruktur Feld werden Daten gleichen Typs zu einer neuen Einheit zusammengefaßt. Die Datenstruktur Verbund kann Daten unterschiedlichen Typs umfassen.

2. Aufgabe
Die Adreßangabe muß bestehen aus:

Verbundvariablen_Name.Komponentenname

3. Aufgabe

```
var Artikel : record
                Nr          : integer;
                Bezeichnung : string (. 25 .);
                Lagermenge  : integer;
                Einzelpreis : real;
              end;
```

4. Aufgabe

```
type ArtSatz =  record
                  Nr          : integer;
                  Bezeichnung : string (. 25 .);
                  Lagermenge  : integer;
                  Einzelpreis : real;
                end;

type ArtFeld = array (.1..50.) of ArtSatz;

var  Artikel : ArtFeld;
```

5. Aufgabe

```
program Fahrplan_Eingabe_Auskunft;
uses CRT;              (* 4 *)
const Max = 30;
type  Zug = record
              Zielort : string (.25.);
              Nr      : string (.8.);
              Abfahrt : real;
            end;
type  Zugfeld = array (. 1.. Max.) of Zug;
var   Fahrplan  : Zugfeld;
      N         : integer;
      Ende      : char;
      Reiseziel : string (.25.);


procedure Bildschirmmaske;
begin
  ClrScr;
  Gotoxy (5,3); writeln ('Zielort:');
  Gotoxy (5,5); writeln ('Zug-Nr :');
  Gotoxy (5,7); writeln ('Abfahrt:');
  Gotoxy (5,9); writeln ('Weiter (j/n):');
end;


procedure Fahrplaneingabe (var Index : integer);
var Weiter : char;
begin
  Bildschirmmaske;
  Index := 0;
  repeat
    Index := Index + 1;
    with Fahrplan (.Index.) do
    begin
      Gotoxy (19,3);ClrEol; readln (Zielort);
      Gotoxy (19,5);ClrEol; readln (Nr);
      Gotoxy (19,7);ClrEol; readln (Abfahrt);
    end;
    Gotoxy (19,9); ClrEol; readln (Weiter);
  until (Weiter in (.'n','N'.)) or (Index = Max);
end;


procedure Anzeige (HoechsterIndex : integer);
var X : integer;
begin
  for X := 1 to HoechsterIndex do
  begin
    with Fahrplan (.X.) do
    if Zielort = Reiseziel
      then writeln (Nr, Abfahrt:6:2);
```

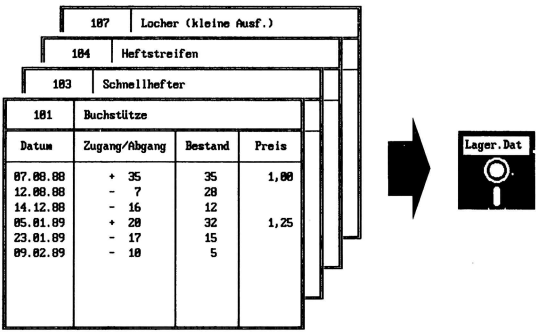
```
  end;
end;


begin                                        (* HAUPTPROGRAMM *)
  Fahrplaneingabe (N);
  repeat
    ClrScr;
    Gotoxy (5,2); write ('Eingabe Reiseziel: ');
    readln (Reiseziel);
    writeln ('Zug-Nr  Abfahrt');
    Anzeige (N);
    writeln;
    writeln ('Ende der Auskunft (j/n)');
    readln (Ende);
  until Ende in ['j','J'];
end.
```

6. Aufgabe

```
type Fach       = record
                    Bezeichnung : string (.20.);
                    Noten       : array (.1..6.) of integer;
                  end;
type Schueler   = record
                    Name: string (.20.);
                    Faecher: array (.1..9.) of Fach;
                  end;
type Klasse     = array (.1..25.) of Schueler;
var  LeistDatei : Klasse;
```

5 DATENSTRUKTUR SEQUENTIELLE DATEI (FILE)

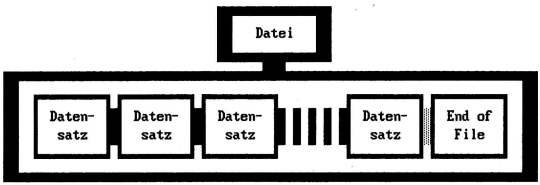
Einführungsbeispiel: Ein Unternehmen möchte seine Lagerwirtschaft rationalisieren. Künftig sollen die Artikeldaten nicht mehr mit einer Lagerkartei, sondern mittels einer Lagerdatei verwaltet werden. Dazu sind die Artikeldaten auf einen externen Datenspeicher zu speichern.



Im Gegensatz zum internen Arbeitsspeicher behalten die externen Speicher die eingegebenen Daten, auch wenn der Rechner abgeschaltet wird. Externe Speicher sind beispielsweise Diskette oder Magnetplatte.
Zum Aufbau und zur Verwaltung von Datenbeständen auf externen Speichern stellt die Programmiersprache Pascal die Datenstruktur Datei (engl. **file**) zur Verfügung.

5.1 Charakteristik einer sequentiellen Datei

Eine Datei besteht aus einer Folge von Sätzen, die meistens die Struktur eines RECORDS haben. Diese Art von Datei wird auch typisierte Datei genannt, weil ihre Datensätze stets denselben Aufbau haben.



Eine Datei besteht aus beliebig vielen Datensätzen. Eine Grenze setzt lediglich das Aufnahmevermögen, die Speicherkapazität des externen Datenträgers.
Das Ende der Datei wird durch ein besonderes Datei-Endemerkmal gekennzeichnet. Es wird **EOF** genannt. EOF ist die Abkürzung für **E**nd **O**f **F**ile (= Ende der Datei). Es wird automatisch hinter den letzten Datensatz gesetzt.
Diese Dateiform wird als **sequentiell** (lat. der Reihe nach) bezeichnet, weil die Datensätze in der Regel nur der Reihe nach bearbeitet werden können. Es gibt innerhalb einer sequentiellen Datei keine Informationen darüber, wo beispielsweise der Datensatz zum Artikel Schnellhefter gespeichert ist. Es kann der erste, im ungünstigsten Fall aber auch der letzte Datensatz sein. Möchte man sich die Angaben zu diesem Artikel anzeigen lassen, so muß die Datei vom Anfang an Satz für Satz durchsucht werden.

5.2. Aufbau und Verarbeitung einer sequentiellen Datei

5.2.1 Grundstrukturen eines Dateiverarbeitungsprogramms

Der Algorithmus zum Aufbau einer Datei läßt sich grob wie folgt beschreiben:

```
Algorithmus Datei_Aufbau
Beginn
  Vereinbaren einer Datei
  Öffnen der Datei
  Wiederhole
    Eingabe des Datensatzes
    Schreiben des Datensatzes in die Datei
  bis Ende der Eingabe
  Schließen der Datei
Ende
```

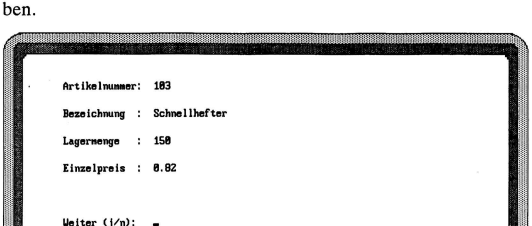
Der Algorithmus zum Lesen einer Datei weist eine ähnliche Struktur auf:

```
Algorithmus Datei_Lesen
Beginn
  Vereinbaren einer Datei
  Öffnen der Datei
  Solange nicht Ende der Datei
    Lesen eines Datensatzes aus der Datei
    Ausgabe des Datensatzes
  Schließen der Datei
Ende
```

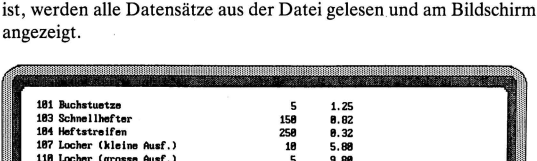
Wie aus den beiden Algorithmen zu ersehen ist, werden bei der Dateiverarbeitung folgende Anweisungen benötigt:

- Vereinbaren
- Öffnen
- Schließen
- Lesen
- Schreiben

Diese Anweisungen werden an Hand des nachstehenden Programms in den folgenden Kapiteln erläutert. Mit dem Programm wird eine Lagerdatei aufgebaut. Dazu werden die Artikeldaten über die nachstehende Bildschirmmaske eingegeben und in die Datei geschrieben.



Nachdem der letzte Datensatz eingegeben und gespeichert worden ist, werden alle Datensätze aus der Datei gelesen und am Bildschirm angezeigt.



Geben Sie das Programm ein und testen Sie es!

```
1  program Lagerdatei_Aufbau_Anzeige;
2  uses CRT;        (* 4 *)
3  type ArtSatz = record
                    Nummer      : integer;
                    Bezeichnung : string (.30.);
                    Menge       : integer;
                    Preis       : real;
                  end;
```

```
 4  var  Artikel    : ArtSatz;
 5       LagerDatei : file of ArtSatz;
 6       Weiter     : char;


 7  procedure Bildschirmmaske;
 8  begin
 9    ClrScr;
10    Gotoxy (5,3); writeln ('Artikelnummer:');
11    Gotoxy (5,5); writeln ('Bezeichnung  :');
12    Gotoxy (5,7); writeln ('Lagermenge   :');
13    Gotoxy (5,9); writeln ('Einzelpreis  :');
14    Gotoxy (5,13); writeln ('Weiter (j/n):');
15  end;


16  procedure Eingabe;
17  begin
18    rewrite (LagerDatei);
19    repeat
20      Gotoxy (21,3); ClrEol; readln (Artikel.Nummer);
21      Gotoxy (21,5); ClrEol; readln (Artikel.Bezeichnung);
22      Gotoxy (21,7); ClrEol; readln (Artikel.Menge);
23      Gotoxy (21,9); ClrEol; readln (Artikel.Preis);
24      write (LagerDatei,Artikel);
25      Gotoxy (21,13); Weiter := ReadKey;        (* 4 *)
                    (* read (kbd, Weiter);        * 3 *)
26    until Weiter in ('n','N');
26    close (LagerDatei);
27  end;


28  procedure Datei_Lesen_Anzeigen;
29  begin
30    reset (LagerDatei);
31    ClrScr;
32    while not eof (LagerDatei) do
33    begin
34      read (LagerDatei, Artikel);
35      with Artikel do
36        writeln (Nummer:6,' ',Bezeichnung,
                  ' ':32-length(Bezeichnung), Menge:10, Preis:10:2);
37    end;
38    close (LagerDatei);
39  end;


40  begin                                  (* HAUPTPROGRAMM *)
41    assign (LagerDatei,'Lager.Dat');
42    Bildschirmmaske;
43    Eingabe;
44    Datei_Lesen_Anzeigen;
45  end.
```

Anmerkung:
In dem Schreibbefehl in Zeile 36

```
36 writeln (Nummer: 6, ' ', Bezeichnung,
            ' ':32-length(Bezeichnung), Menge: 10, Preis: 10:2);
```

wird durch **' ':32-length(Bezeichnung)** erreicht, daß die nachfolgende Ausgabe von Menge und Preis spaltengerecht erfolgt. Es werden nach der Ausgabe der Stringvariablen so viele Leerstellen ausgegeben, wie die Differenz von 32 abzüglich der effektiven Länge der Zeichenkette angibt.
Beispiel:
In der Variablen Bezeichnung steht die Zeichenkette "Schlüssel". Die effektive Länge von "Schlüssel" ist 8. Es würden dann 32 − 8 = 24 Leerstellen im Anschluß an "Schlüssel" ausgegeben.

5.2.2 Vereinbaren einer Datei

Um eine Datei zu vereinbaren, sind zwei Schritte erforderlich. Zum einen ist eine Dateivariable zu vereinbaren. Zum anderen ist im Programm zu vermerken, unter welchem Namen die Datei auf dem externen Datenspeicher geführt werden soll.
Die **Dateivariable** wird im Vereinbarungsteil des Programms mit Hilfe des Schlüsselwortes **FILE** festgelegt.

```
5  LagerDatei: file of ArtSatz;
```

Hinter **of** wird angegeben, von welchem Datentyp die Komponenten der Datei sind. In unserem Beispiel ist es der Datentyp ArtSatz, der in Zeile 3 als RECORD festgelegt wurde.
Allgemein lautet die Anweisung zur Vereinbarung einer Dateivariablen:

var Dateivariable: file of Datentyp;

Die so vereinbarte Dateivariable wird häufig auch als **logischer** Dateiname bezeichnet. Dadurch wird auf den Unterschied zum Namen der physischen Datei hingewiesen. Der **"physische"** Name ist der Name, unter dem die Datei auf dem externen Speicher verwaltet wird. Wenn Sie das obige Programm eingeben und testen, wird auf dem externen Datenträger eine Datei mit dem Namen Lager.Dat erzeugt. Sie können sich davon mit Hilfe des Dir-Kommandos überzeugen.

LagerDatei

ASSIGN

001	101	Buchstuetze	5	1.25
002	103	Schnellhefter	150	0.82
003	104	Heftstreifen	250	0.32
004	107	Locher (kleine Ausf.)	10	5.80
005	110	Locher (grosse Ausf.)	5	9.80
006	115	Ordner (DIN A4)	15	3.54

Lager.Dat

101	Buchstuetze	5	1.25	EOL
103	Schnellhefter	150	0.82	EOL
104	Heftstreifen	250	0.32	EOL
107	Locher (kleine Ausf.)	10	5.80	EOL
110	Locher (grosse Ausf.)	5	9.80	EOL
115	Ordner (DIN A4)	15	3.54	EOL

EOF

Die Zuordnung zwischen logischem und physischem Dateinamen erfolgt in der **ASSIGN**-Anweisung:

```
41  assign (LagerDatei, 'Lager.Dat');
```

Mit der ASSIGN-Anweisung (engl.: zuweisen) wird dem System mitgeteilt, daß sich die Dateivariable LagerDatei auf die physische Datei mit dem Namen Lager.Dat beziehen soll. Der ASSIGN-Befehl ist die einzige Stelle im Programm, an der der physische Dateiname aufgeführt wird. In allen anderen Dateianweisungen wird die Datei stets nur mit dem logischen Dateinamen angesprochen.
In allgemeiner Form lautet die Zuordnungsanweisung:

assign (Dateivariable, 'Laufwerk: Physischer_Dateiname');

Unter Laufwerk ist anzugeben, in welchem Laufwerk sich der Datenträger mit der betreffenden Datei befindet. Soll die Lagerdatei auf der Diskette im Laufwerk b: gespeichert werden, so müßte die ASSIGN-Anweisung lauten:

```
assign (LagerDatei, 'b:Lager.Dat');                    (1)
```

Liegt die Datei in einem Unterverzeichnis, dann muß der Zugriffspfad im ASSIGN-Befehl beschrieben werden.

Beispiel: `assign (LagerDatei, 'b:\Verwa\Lager.Dat');`

In dem obigen Programm wurde die physische Datei durch eine Stringkonstante angegeben. Es ist aber auch möglich, die physische Datei in einer String-Variablen zu benennen.

Beispiel:
```
write ('Wie soll die Datei heißen?');
readln (Name_Phys_Datei);
assign (LagerDatei, Name_Phys_Datei);                  (2)
```

Wird in der Leseanweisung die Zeichenfolge

`b:\Verwa\Lager.Dat.`

eingetastet, so hat die ASSIGN-Anweisung (2) dieselbe Wirkung wie die Zuordnungsanweisung (1).

5.2.3 Öffnen einer Datei

Die Anweisungen zum Öffnen einer Datei lauten **REWRITE** bzw. **RESET**. Mit den Öffnungsanweisungen werden eine Reihe von Betriebssystemroutinen ausgelöst, die für einen sicheren Datenverkehr zwischen Arbeitsspeicher und externem Datenspeicher sorgen. Das ist aber auf dieser Ebene der Programmierung nicht weiter wichtig. In diesem Zusammenhang interessiert uns vor allem die Wirkung dieser beiden Befehle.

REWRITE
Mit der Anweisung

```
18 rewrite (LagerDatei);
```

wird aufgrund der vorausgehenden ASSIGN-Anweisung auf dem externen Datenträger eine Datei unter dem Namen Lager.Dat eingerichtet. Der **Satzzeiger**, der auf den aktuellen Datensatz der physischen Datei zeigt, wird auf den Anfang der Datei gesetzt.
Die Datei ist damit zur Aufnahme von Datensätzen vorbereitet.
Die allgemeine Form der REWRITE-Anweisung lautet:

rewrite (Dateivariable);

Bei der Programmgestaltung ist zu beachten, daß der REWRITE-Befehl eine Nebenwirkung hat, die bisweilen recht unangenehm sein kann: Nehmen wir einmal an, Sie hätten mit dem obigen Programm 100 Artikelsätze erfaßt und sich die Artikeldaten am Bildschirm anzeigen lassen. Wenn Sie nun das Programm erneut aufrufen und starten, wird Ihre mühsam aufgebaute Lagerdatei durch die REWRITE-Anweisung **ohne Vorwarnung** gelöscht.

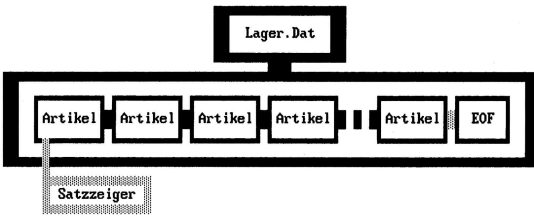
> Stimmt ein Dateiname auf dem externen Datenträger genau mit dem in der ASSIGN-Anweisung festgelegten physischen Dateinamen überein, so wird die schon vorhandene Datei auf dem Datenträger gelöscht.

RESET

Der Datei-Öffnungsbefehl RESET ist anzuwenden, wenn eine Datei bereits eingerichtet ist. Mit

```
30  reset (LagerDatei);
```

wird die Datei Lager.Dat zur Verarbeitung geöffnet. Der Satzzeiger wird auf den 1. Datensatz positioniert.



In der Datei kann nun sowohl gelesen wie auch geschrieben werden.
Die allgemeine Form des Befehls zum Öffnen einer bereits bestehenden Datei lautet:

reset (Dateivariable);

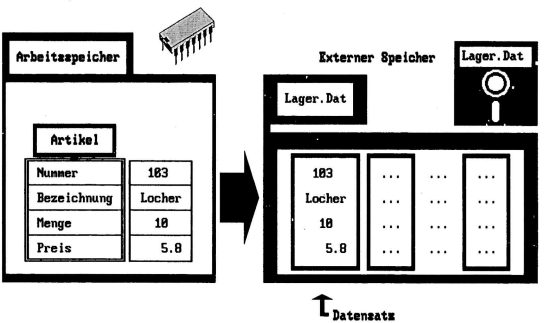
Falls die der Dateivariablen zugeordnete physische Datei nicht existiert, wird das Programm mit einer Fehlermeldung abgebrochen, sofern dagegen keine besonderen Vorkehrungen getroffen worden sind.
Wird auf eine bereits geöffnete Datei erneut eine RESET-Anweisung gegeben, so hat das keine negativen Nebenwirkungen. Die Datei wird erst geschlossen und anschließend wieder geöffnet. Der Satzzeiger steht dann wieder auf dem Beginn des ersten Datensatzes.

5.2.4 Schreiben in eine Datei

Die Vorbereitungen zum Einrichten und Öffnen einer Datei sind zwar ein wenig umständlich. Dafür ist aber in Turbo-Pascal das Schreiben in eine Datei leicht zu programmieren. Ein Datei-Schreibbefehl ist in Zeile 24 dargestellt:

24 write (LagerDatei,Artikel);

Mit dieser Anweisung wird der Inhalt der Variablen Artikel mit all seinen Bestandteilen in die Datei geschrieben. Da die Daten auf dem externen Speichermedium in der gleichen binären Form wie im Arbeitsspeicher dargestellt werden, handelt es sich bei dieser Schreibanweisung um eine Art Kopierbefehl.



In der Datei-Schreibanweisung wird als erste Angabe in der runden Klammer die Dateivariable, die auf die Zieldatei verweist, angegeben. Darauf folgt der Name der Variablen, deren Inhalt in die Zieldatei geschrieben werden soll.
Die allgemeine Form der WRITE-Anweisung zum Schreiben in eine Datei lautet also:

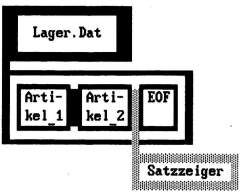
WRITE (Dateivariable, Variablenname)

Voraussetzung für die Schreibanweisung ist, daß der Datentyp der Dateikomponenten **identisch** ist mit dem Datentyp der Variablen, deren Inhalt in die Datei geschrieben wird. Beide sind mit derselben Typangabe zu vereinbaren, wie aus dem obigen Programm zu ersehen ist.

```
4 var Artikel: ArtSatz:
5       Lager-Datei: file of ArtSatz;
```

Der ausgegebene Datensatz wird in der physischen Datei an die Stelle geschrieben, auf die der Satzzeiger zeigt. Nach Abschluß des Speicherbefehls steht der Satzzeiger hinter dem zuletzt abgelegten Satz. Das EOF-Merkmal wird vom System automatisch mitgeführt.

Beim nächsten Befehl zum Speichern wird der Datensatz hinter dem vorher geschriebenen Satz angehängt.

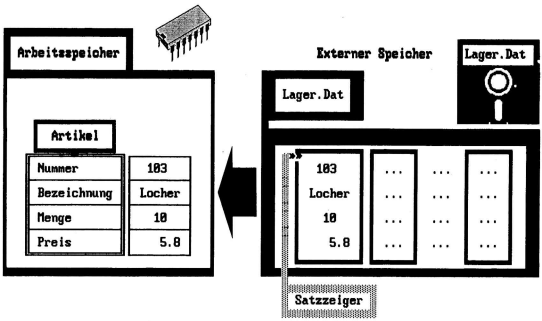


5.2.5 Lesen aus einer Datei

Die Anweisung zum Lesen eines Datensatzes aus einer Datei hat den gleichen Aufbau wie die Schreibanweisung. Statt write heißt es nun **read**.

```
34  read (LagerDatei, Artikel);
```

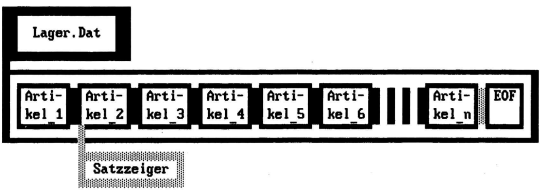
Mit dieser Anweisung wird der Datensatz, auf dessen Anfang der Satzzeiger zeigt, in den Arbeitsspeicher gelesen.



Da es sich hier auch um eine Art Kopiervorgang handelt, brauchen die einzelnen Komponenten des Datensatzes nicht angegeben zu werden. Der Inhalt des Datensatzes wird als eine Einheit in die Verbundvariable übertragen.
Allgemein lautet die Anweisung zum Lesen eines Datensatzes aus einer Datei

read (Dateivariable, Variablenname)

Natürlich muß auch hier wie bei der Schreibanweisung die Typengleichheit zwischen den Komponenten der Datei und der aufnehmenden Variablen bestehen.
Nach dem Lesevorgang steht der Satzzeiger auf dem Beginn des nächsten Datensatzes. Bei jedem Lesevorgang wird der Satzzeiger um einen Datensatz weitergesetzt.



Diese Art des Lesens bezeichnet man als Reihenfolgelesen bzw. sequentielles Lesen. Damit beim sequentiellen Lesen nicht über das Datei-Ende hinausgegriffen wird, prüft das System bei jedem Lesevorgang, ob die EOF-Marke erreicht wurde. Falls nach Erreichen der EOF-Marke weitere Lesebefehle gegeben werden, bricht das Programm mit der Fehlermeldung "Platten-Lese-Fehler" ab.

5.2.6 Schließen einer Datei

Nach Beendigung der Schreib- und Leseoperationen auf der Datei ist die Datei durch den CLOSE-Befehl zu schließen:

```
38  close (LagerDatei);
```

Durch diese Anweisung wird die physische Datei Lager.Dat geschlossen. Es wird also auch beim Schließen der Datei der logische Name der Datei verwendet. In allgemeiner Form lautet die CLOSE-Anweisung:

close (Dateivariable)

Mit CLOSE werden Ausgabedaten, die sich noch in einem internen Schreibpuffer befinden, in die Datei eingetragen und der Directory-Eintrag auf den neuesten Stand gebracht.
Falls Sie in einem Programm das CLOSE vergessen haben, kann es Ihnen passieren, daß eingegebene Daten verlorengehen.
Anmerkung: Im Gegensatz zur Pascal-Version 3.0 führt in der Version 4.0 ein Close auf eine nicht geöffnete Datei zu einem Laufzeitfehler.

5.2.7 Zusammenfassung

- Eine **Datei (File)** ist eine Folge von Komponenten mit gleichem Datentyp. Die Anzahl der Komponenten ist variabel. Sie wird begrenzt durch die Kapazität des externen Speichers, auf dem die Datei aufgebaut wird.
- Das Ende einer Datei wird durch eine **EOF**-Marke kenntlich gemacht, die automatisch vom System gesetzt wird (EOF = End Of File).
- Es ist zu unterscheiden zwischen der **Dateivariablen**, unter der die Datei im Programm angesprochen wird, und dem **Namen der phy-**

sischen Datei auf dem externen Speicher, unter dem das Betriebssystem auf die Datei zugreift.

- Die Dateivariable wird vereinbart durch

  Dateivariable: **file** of Datentyp

- Die Zuordnung zwischen der Dateivariablen und der physischen Datei erfolgt durch die ASSIGN-Prozedur:

  assign (Dateivariable, Name_physische_Datei)

- Der Name der physischen Datei kann in einer String-Konstanten oder in einer String-Variablen angegeben werden.
- Eine Datei wird geöffnet durch

  > **rewrite** (Dateivariable), wenn die Datei neu eingerichtet werden soll.
  > **reset** (Dateivariable), wenn es sich um eine bereits bestehende Datei handelt.

- Eine Datei wird geschlossen mittels

  close (Dateivariable)

- Ein Datensatz wird mittels

  write (Dateivariable, Variablenname)

  in die Datei geschrieben. Die Datei und die Variable müssen denselben Datentyp aufweisen.

- Ein Datensatz wird mittels

  read (Dateivariable, Variablenname)

  aus einer Datei in eine Variable gelesen. Die Variable und die Datei müssen vom selben Datentyp sein.

5.2.8 Übungen

1. Aufgabe
Welcher Hauptunterschied besteht zwischen der Datenstruktur Feld und der Datenstruktur Datei?

2. Aufgabe
Wozu dient die ASSIGN-Prozedur?

3. Aufgabe
Der Datensatz für einen Arbeitnehmer besteht aus einem Verbund:

```
type Arbeitnehmer = record Nummer: integer;
                           Name: string (.40.);
                           Geburtstag: string (.8.);
                           Steuerklasse, Kinderzahl,
                           Tarifstufe: integer;
                    end;
```

Wie lautet die Vereinbarung der Dateivariablen?

4. Aufgabe
Die Mitarbeiterdatei steht auf der Diskette im Laufwerk b:. Ihr Dateiname sei ArbNehm.Dat. Wie erfolgt die Zuordnung zwischen der Dateivariablen und der physischen Datei?

5. Aufgabe
Wie wird die Mitarbeiterdatei geöffnet, wenn die Datei bereits vorhanden ist und mit ihr weitergearbeitet werden soll?

5.2.9 Lösungen

1. Aufgabe
Bei der Datenstruktur Feld muß im Programm die Anzahl der Komponenten fest angegeben werden. Bei der Datenstruktur Datei entfällt diese Angabe, weil die Anzahl der Komponenten variabel ist.

2. Aufgabe
Mittels der ASSIGN-Prozedur wird der Dateivariablen eine externe Datei zugeordnet.

3. Aufgabe
Die Vereinbarung der Dateivariablen könnte lauten:

```
var Mitarbeiterdatei: file of Arbeitnehmer;
```

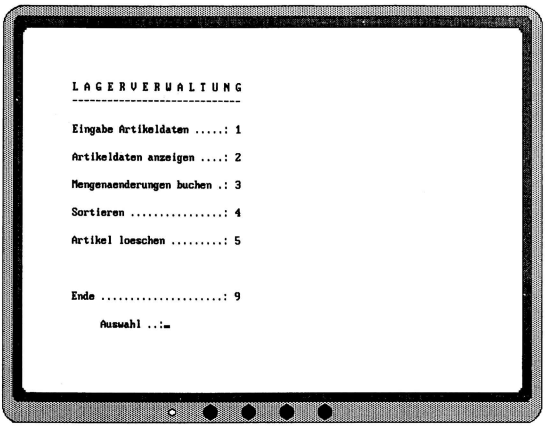
4. Aufgabe
Die Zuordnung erfolgt über

```
assign (Mitarbeiterdatei, 'b:ArbNehm.Dat');
```

5. Aufgabe
Die Datei wird mittels reset (Mitarbeiterdatei) geöffnet.

5.3 Algorithmen zur Dateibearbeitung

Für die Dateibearbeitung müssen in einem Dateiverwaltungsprogramm eine Vielzahl von Prozeduren bereitgestellt werden. Die wichtigsten werden in diesem Kapitel behandelt. Aus den Prozeduren ergibt sich schließlich ein komplettes Lagerverwaltungsprogramm. Sein Funktionsumfang ist aus der nachstehenden Abbildung des Auswahlbildschirms zu ersehen.



Hinweis:
Damit Sie die Programmentwicklung am Rechner nachvollziehen können, geben Sie die Variablenvereinbarung, die Prozedur-Auswahl und das Hauptprogramm ein. Diese Programmteile sind auf Seite 150/151 dargelegt. Entsprechend dem Lernfortschritt fügen Sie die behandelten Prozeduren in das "Rahmenprogramm" ein. Auf diese Weise komplettieren Sie das Programm.

Damit bei den Prozeduraufrufen auch die Dateivariable als Variablenparameter jeweils übergeben werden kann, wird gegenüber dem Vereinbarungsteil des Programms Lagerdatei_Aufbau_Anzeige folgende Änderung vorgenommen:

```
Type ...
     LagerFile  = file of ArtSatz;
var  Artikel    : ArtSatz;
     LagerDatei : LagerFile;
     Wahl       : char;
     Ende       : boolean;
```

5.3.1 Allgemeine Datei-Öffnungsprozedur

Um bei den Arbeiten mit einer Datei nicht Gefahr zu laufen, eine bereits bestehende Datei aus Versehen zu löschen, wird das Öffnen der Datei in einer Öffnungsprozedur durchgeführt. In ihr wird eine Datei nur dann durch REWRITE geöffnet, wenn sie mit RESET nicht geöffnet werden konnte.

```
1  procedure DateiOeffnen (var LagDatei: LagerFile);
2  begin
3    {*$I-*}
4    reset (LagDatei);
5    if IORESULT <> 0
6      then rewrite (LagDatei);
7    if IORESULT <> 0
8      then begin
9        writeln; write ('Datei kann nicht geoeffnet werden');
10       Delay (1000); halt;
11     end;
12   {*$I+*}
13 end;
```

Die Prozedur wird aufgerufen durch

```
DateiOeffnen (LagerDatei);
```

Erläuterungen
Im "Normalzustand" wird ein Programmablauf automatisch abgebrochen, wenn ein Fehler bei einer Eingabe-/Ausgabeoperation aufgetreten ist. Diese Automatik läßt sich durch die Compileranweisung **(*$I−*)** ausschalten.
In dieser Prozedur wurde in Zeile 3 die Compileranweisung **(*$I−*)** gegeben. Durch den RESET-Befehl wird zunächst versucht, die Datei zu öffnen. Wenn sie nicht existiert, vermerkt das System den Fehler in einer internen Variablen, die mittels der Funktion **IORESULT** abgefragt werden kann, bricht aber nicht selbsttätig das Programm ab.
Mit Hilfe der Funktion **IORESULT** (= Input-Output-RESULT) wird geprüft, ob die Dateioperation korrekt ausgeführt worden ist.

```
5 if IORESULT < >0
```

Liefert IORESULT den Wert 0, so liegt kein Fehler bei einer Ein-/Ausgabeoperation vor. Die Datei konnte also durch ein RESET geöffnet werden.
Erbringt IOResult jedoch einen ganzzahligen Wert > 0, so wurde die RESET-Anweisung nicht erfolgreich ausgeführt, weil die Datei noch nicht bestand. Somit darf jetzt die Datei mit REWRITE geöffnet werden, ohne dabei Gefahr zu laufen, eine bereits existierende Datei zu zerstören. Wenn nach der REWRITE-Anweisung IORESULT wiederum ungleich Null ist, so herrscht auf dem externen Datenträger wahrscheinlich Platzmangel.
Vor dem Verlassen der Prozedur ist mit **(*$I+*)** die Abbruchautomatik wieder einzuschalten.

5.3.2 Suchen in einer Datei

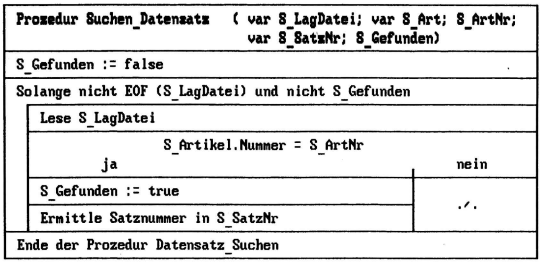
Bei der Eingabe von Artikelsätzen passiert es gelegentlich, daß bei unterschiedlichen Datensätzen versehentlich dieselbe Artikelnummer eingegeben und gespeichert wird. Das darf nicht sein! Um das Programm gegen solche Eingabefehler abzusichern, muß eine Suchprozedur im Programm vorhanden sein, die die Datei nach Artikel-

nummern durchsucht. Da auch bei anderen Funktionen des Lagerverwaltungsprogramms Artikelsätze mit bestimmten Artikelnummern gesucht werden müssen, wird die Suchprozedur in der Weise aufgebaut, daß sie auch aus anderen Prozeduren aufgerufen werden kann.
Die Suchprozedur soll für die aufrufende Prozedur ermitteln, ob zu der gesuchten Artikelnummer bereits ein Artikelsatz in der Datei besteht. Falls ja, sollen auch die Satznummer und die Daten des Artikelsatzes für die rufende Prozedur bereitgestellt werden. Dazu ist erforderlich, daß die entsprechenden Variablen als **Variablenparameter** an die Prozedur Suchen übergeben werden.
Als formale Parameter werden festgelegt:

Name	**Inhalt**
var S_LagDatei: LagerFile;	Dateivariable
var S_Art: ArtSatz;	Artikelsatz
S_ArtNr: integer;	Suchnummer
var S_Art: Longint;	Nummer des Artikelsatzes
(* 3: integer;*)	in der Datei
var S_Gefunden: boolean);	true, wenn Suchnummer in Datei gefunden, sonst false

Die zu suchende Artikelnummer wird als Werteparameter auf S_ArtNr übertragen, da sie nicht an die rufende Prozedur zurückgegeben zu werden braucht.
Um sich die Bezeichnung der Parameter zu vereinfachen, wird den Parametern der Anfangsbuchstabe des Prozedurnamens vorangesetzt.
Ablauf der Suchprozedur: Die Datei wird Satz für Satz gelesen. Bei jedem Satz wird geprüft, ob die eingelesene Artikelnummer gleich der gesuchten Artikelnummer ist, die von der aufrufenden Prozedur übergeben wurde. Bei Gleichheit wird die boolsche Variable Gefunden auf wahr gesetzt und die Satznummer ermittelt.
Die Suchschleife wird durchgeführt, solange nicht das Dateiende erreicht wurde und die Variable Gefunden nicht den Wert true aufweist.

Algorithmus



Programm

```
1  procedure Suchen_Datensatz (var S_LagDatei : LagerFile ;
                               var S_Art      : ArtSatz;
                                   S_ArtNr    : integer;
                               var S_SatzNr   : LongInt; (* 4 *)
                                              (* integer;  * 3 *)
                               var S_Gefunden : boolean);
2  begin
3    DateiOeffnen (S_LagDatei);
4    S_Gefunden := false;
5    while ( not eof (S_LagDatei)) and (not S_Gefunden) do
6    begin
7      read (S_LagDatei, S_Art);
8      if S_Art.Nummer = S_ArtNr
9        then begin
10         S_Gefunden := true;
11         S_SatzNr := filepos (S_LagDatei);
12       end;
13   end;
14 end;
```

Entsprechend der im Prozedurkopf festgelegten formalen Parameter muß der Prozeduraufruf folgenden Aufbau haben:

Suchen_Datensatz (B_LagDatei, B_Artikel,
B_SatzNr, B_Gefunden);

Erläuterungen
Bei jedem Aufruf der Prozedur Suchen wird die Prozedur DateiOeffnen angesprochen. Das ist erforderlich, damit der Satzzeiger wieder auf den Dateianfang gestellt wird.

Vor einem sequentiellen Lesen der Datei muß der Satzzeiger auf den Anfang der Datei positioniert werden.

Mittels der Funktion **FILEPOS**

```
11   S_SatzNr:= filepos (S_LagDatei);
```

wird in der Variablen S_SatzNr festgehalten, auf dem wievielten Datensatz der Satzzeiger gerade steht.
Allgemein lautet die Anwendung der FILEPOS-Funktion:

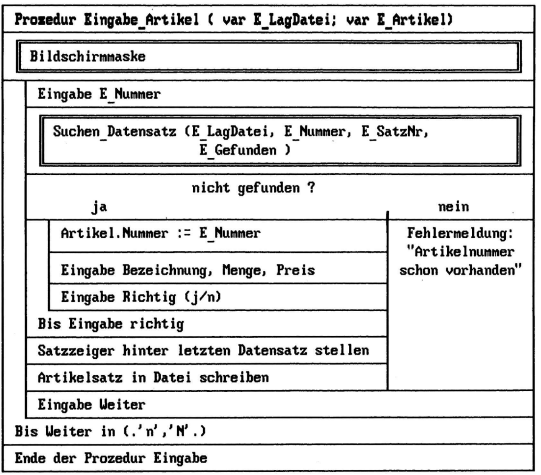
Satznummer:= filepos (Dateivariable)

In Satznummer wird die aktuelle Position des Satzzeigers vermerkt. Steht der Satzzeiger auf dem ersten Datensatz, so bildet sich in Satznummer der Wert 1.
Satznummer ist vom Typ INTEGER (*3*) oder vom Typ LongInt (*4*).

5.3.3 Anfügen von Datensätzen

Das obige Programm (siehe Abschnitt 5.2.1) hat den Nachteil, daß es bei einem erneuten Start die neu eingegebenen Datensätze an den Dateianfang schreibt. Dadurch werden bisher gespeicherte Datensätze zerstört. Dieser Mangel wird in der nun zu entwickelnden Eingabeprozedur behoben. Jeder neue Datensatz wird stets an das Ende der Datei angefügt. Ferner soll in der Prozedur festgestellt werden, ob eine eingegebene Artikelnummer bereits in der Datei existiert. Dazu können wir die Prozedur Suchen_Datensatz einsetzen.
Der Arbeitsablauf in der Prozedur ist wie folgt zu steuern: Nach der Eingabe der Artikelnummer wird zunächst geprüft, ob die Artikelnummer bereits in der Datei vorhanden ist. Ist das der Fall, erscheint ein entsprechender Hinweis am Bildschirm. Andernfalls werden die Artikeldaten eingegeben. Deren Korrektheit ist zu bestätigen, bevor die Daten gespeichert werden. Falls die Eingabe Fehler aufweist, werden die Artikeldaten erneut eingegeben. Nach jeder Speicherung wird gefragt, ob die Dateneingabe fortgesetzt werden soll.

Algorithmus



Programm

```
1  procedure Eingabe_Artikel (var E_LagDatei : LagerFile;
                              var E_Artikel  : ArtSatz);
2  var E_Nummer              : integer;
       E_SatzNr, SatzAnzahl : longint; (* 4 *)
                             (* integer); * 3 *)
       E_Gefunden           : boolean;
       Richtig, Weiter      : char;
3  begin
4    Bildschirmmaske;
5    repeat
6      Gotoxy (21,3); ClrEol; readln (E_Nummer);
7      Suchen_Datensatz (E_LagDatei, E_Artikel, E_Nummer,
8                        E_SatzNr,E_Gefunden);
9    if not E_Gefunden
10     then begin
11       with E_Artikel do
12       begin
13         Nummer := E_Nummer;
14         repeat
15           Gotoxy (21,5); ClrEol; readln (Bezeichnung);
16           Gotoxy (21,7); ClrEol; readln (Menge);
17           Gotoxy (21,9); ClrEol; readln (Preis);
18           Gotoxy (5,13); write ('Richtig (j/n)');
19           Richtig := ReadKey; (* 4 *)
           (* read (kbd, Richtig); * 3 *)
20           DelLine;
21         until Richtig in ('j','J');
22       end; (* with *)
23       SatzAnzahl := filesize (E_LagDatei);
24       seek (E_LagDatei,SatzAnzahl );
25       write (E_LagDatei,E_Artikel);
26     end
27     else begin
29       Gotoxy (10,20);
30       write ('Artikelnummer bereits vorhanden <RETURN>');
31       Weiter := ReadKey; (* 4 *)
       (* read (kbd, Weiter); * 3 *)
32       DelLine;
33     end;
34     Gotoxy (5,13); write ('weiter ? (j/n)');
35     Weiter := ReadKey; (* 4 *)
     (* read (kbd, Weiter); * 3 *)
36     DelLine;
37   until Weiter in ('n','N');
38 end;
```

Die Prozedur Eingabe_Artikel wird aus dem Hauptprogramm aufgerufen durch

```
Eingabe_Artikel (LagerDatei, Artikel);
```

Erläuterungen

Ein neuer Datensatz kann nur dann eingegeben werden, wenn die boolsche Variable E_Gefunden auf false steht (Zeile 9). Der Wert für E_Gefunden wurde in der Prozedur Suchen_Datensatz, an die E_Gefunden als Variablenparameter übergeben wurde, ermittelt.
In dem Prozeduraufruf Suchen_Datensatz

```
7  Suchen_Datensatz (E_LagDatei, E_Artikel, E_Nummer,
8                    E_SatzNr, E_Gefunden);
```

sind – bezogen auf die Prozedur Eingabe – die Parameter E_Artikel und E_SatzNr eigentlich nicht notwendig. Sie müssen aber als aktuelle Parameter aufgeführt sein, weil im Kopf der Prozedur Suchen_Datensatz entsprechende formale Parameter vereinbart sind.
Neue Datensätze können immer nur an das Ende der Datei vor dem EOF-Merkmal angefügt werden. Deshalb muß vor dem Eintrag eines neuen Datensatzes in die Datei der Satzzeiger an das Ende der Datei gestellt werden. Dazu sind zwei Operationen erforderlich.
Zunächst muß festgestellt werden, wie viele Datensätze in der Datei bereits enthalten sind. Dazu bietet Turbo-Pascal die Standardfunktion

filesize (Dateivariable)

an. Die Funktion FILESIZE (engl. Dateiumfang) ermittelt die Anzahl der Datensätze, die in der Datei enthalten sind. Weist die Datei noch keinen Eintrag auf, so liefert FILESIZE den Wert Null.
In dem obigen Programm wurde die Satzanzahl ermittelt durch:

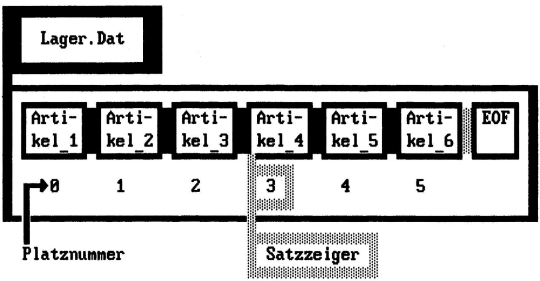
```
23  SatzAnzahl := filesize (E_LagDatei);
```

Die Variable SatzAnzahl muß vom Typ INTEGER (*3*) bzw. vom Typ LongInt (*4*) sein.
Die Information, wie viele Datensätze bereits in die Datei eingetragen worden sind, wird für die zweite Operation benötigt, in der der Satzzeiger hinter den letzten Datensatz positioniert wird. Um den

Satzzeiger auf einen bestimmten Datensatz zu stellen, wenden wir die Standard-Prozedur **SEEK** an. Beispielsweise wird durch

```
seek (LagerDatei,3)
```

der Satzzeiger auf den Datensatz mit der **Platznummer** 3 gesetzt. Da die Zählweise der Platznummer mit Null beginnt, weist der Satzzeiger auf den 4. Dateneintrag.



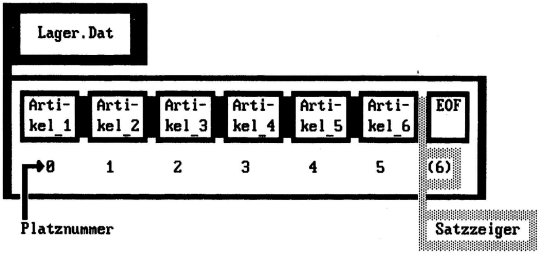
Wird nun, bezogen auf die voranstehende Abbildung, durch

```
SatzAnzahl := filesize (LagerDatei)
```

die Anzahl der Dateneinträge ermittelt, so beinhaltet SatzAnzahl nun den Wert 6. Durch

```
seek (LagerDatei, SatzAnzahl)
```

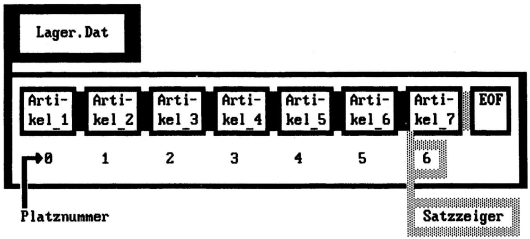
wird der Satzzeiger auf den Beginn des Datensatzes mit der Platznummer 6 gesetzt (siehe Seite 135):



Bei dem anschließenden Schreibbefehl

```
25   write (E_LageDatei, E_Artikel);
```

wird der aktuelle Artikelsatz hinter Artikel_6 angefügt und die EOF-Marke verschoben.



Allgemein lautet die Prozedur, mit der der Satzzeiger auf einen bestimmten Datensatz gestellt werden kann,

seek (Dateivariable, Platznummer)

Platznummer muß eine INTEGER-Konstante oder eine INTEGER-Variable (*3*) bzw. LongInt-Variable (*4*) sein. Es ist zu

beachten, daß die Zählweise für die Platznummer mit Null beginnt. Die Platznummer des ersten Dateneintrags ist also Null.
Um den Satzzeiger auf das Datei-Ende zu stellen, kann man auch die Funktion FILESIZE in die SEEK-Prozedur einbeziehen. Mit

seek (Dateivariable, **filesize(Dateivariable)**)

wird der Satzzeiger hinter den letzten Dateneintrag positioniert.

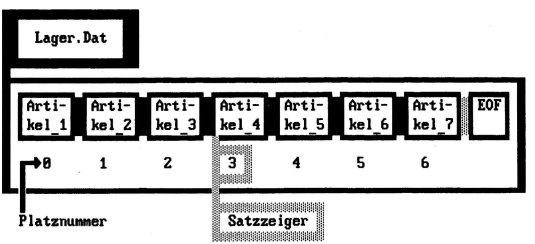
5.3.4 Ändern von Datensätzen

Zu einem Dateiverwaltungsprogramm gehört auch eine Prozedur, mit der gespeicherte Datensätze verändert werden können. Die Programmschritte, die für das Ändern von Datensätzen erforderlich sind, werden am Beispiel der Lagerbestandsveränderung behandelt. Sie werden in der Prozedur Buchen zusammengefaßt.

Der Bedienungsablauf wird in der Prozedur so gestaltet, daß der Sachbearbeiter die Artikelnummer eingibt. Existiert die Artikelnummer, wird der Datensatz angezeigt. Der Bediener gibt daraufhin die Bestandsveränderung ein. Bei einer Lagerentnahme wird die Zahl negativ eingegeben. Der geänderte Lagerbestand wird angezeigt und der Datensatz in die Datei zurückgeschrieben.
Falls die Artikelnummer nicht gefunden wird, meldet der Rechner: "Artikelnummer nicht vorhanden".
Die Buchungen können so lange durchgeführt werden, bis die Frage "Weiter (j/n)" mit n(ein) beantwortet wird.
Grobstruktur des Algorithmus:

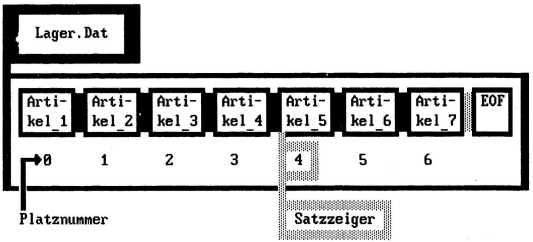
```
Algorithmus Buchen
Beginn
Wiederhole
  Eingabe ArtikelNummer
    Artikelnummer in der Datei suchen
    wenn Artikelnummer gefunden
          dann   Anzeige Artikeldaten
                 Eingabe Mengenänderung
                 Menge := Menge + Mengenänderung
                 Artikelsatz zurückschreiben
          sonst  Anzeige Fehlermeldung
  Eingabe Weiter
Bis Weiter = 'n' oder 'N'
Ende
```

Bei dieser Aufgabenstellung tritt ein Problem auf, das mit den bisherigen Kenntnissen noch nicht gelöst werden kann. Nach der Eingabe der Mengenänderung ist der Artikelsatz auf dieselbe Stelle zurückzuschreiben, auf der er vorher gestanden hat. Würde der Artikelsatz nach der Berechnung der neuen Bestandsmenge unmittelbar in die Datei zurückgeschrieben, so würde der folgende Datensatz überschrieben.



Stellung des Satzzeigers vor dem Lesen

Enthält beispielsweise der Artikel_4 die gesuchte Artikelnummer und ist der Artikel_4 gelesen worden, so steht der Satzzeiger am Anfang des Datensatzes von Artikel_5.



Stellung des Satzzeigers nach dem Lesen

Würde der Artikel_4 nun zurückgeschrieben, so würde der Platz, auf dem bislang Artikel_5 steht, mit den geänderten Daten von Artikel_4 überschrieben. Artikel_4 stünde nun zweimal in der Datei, und Artikel_5 wäre gelöscht.
Um ein Überschreiben des nächsten Datensatzes zu verhindern, muß also vor dem Zurückschreiben eines Datensatzes der Satzzeiger zurückgesetzt werden. Dazu sind zwei Schritte erforderlich.

1. Schritt: Es muß festgestellt werden, an wievielter Stelle der zu bearbeitende Artikelsatz steht.
2. Schritt: Vor dem Zurückschreiben muß der Satzzeiger auf den Anfang des entsprechenden Datensatzes positioniert werden.

Die aktuelle Position des Satzzeigers wird, wie bereits bei der Behandlung der Prozedur Suchen_Datensatz dargelegt wurde, über die Funktion FILEPOS ermittelt. In unserem Beispiel würde die Anweisung

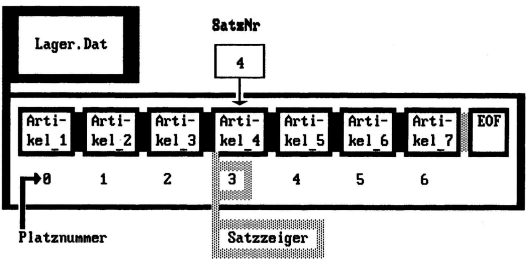
```
SatzNr := filepos (Lagerdatei);
```

in der Variablen SatzNr den Wert 4 bilden. Da die Satznummer in der Prozedur Suchen_Datensatz bereits ermittelt und der aufrufenden Prozedur bereitgestellt wird, braucht in der Prozedur Buchen die Nummer des aktuellen Satzes nicht mehr bestimmt zu werden.
Zum Positionieren des Satzzeigers verwenden wir die SEEK-Prozedur. Dabei ist jedoch zu beachten, daß sich SEEK auf die **Platznum-**

mer bezieht, deren Zählweise mit Null beginnt. Da die Satznummer also stets um 1 höher ist als die Platznummer, muß die SEEK-Anweisung wie folgt formuliert werden:

SEEK (LagerDatei, SatzNr – 1)

Wenn SatzNr den Wert 4 enthält, wird der Satzzeiger durch die obige SEEK-Anweisung auf den Satz mit der Platznummer 3 gestellt. Damit steht der Satzzeiger auf dem Beginn von Artikel_4.



Stellung des Satzzeigers nach SEEK (LagerDatei, SatzNr - 1)

Aus diesen Überlegungen ergibt sich folgende Prozedur zum Buchen von Lagerbestandsveränderungen:

Programm

```
1  procedure Buchen (var B_LagDatei : LagerFile;
                     B_Artikel     : ArtSatz);
2  var B_ArtNr, MengenAenderung : integer;
       B_Gefunden              : boolean;
       B_SatzNr                :longint; (* 4 *)
                               (* integer;  * 3 *)
       Weiter                  : char;


3  procedure B_Bildschirm;
4  begin
5    ClrScr;
6    Gotoxy (5, 1);
     writeln ('LAGERVERWALTUNG: BUCHEN MENGENAENDERUNG');
7    Gotoxy (5, 3); writeln ('Artikelnummer:');
8    Gotoxy (5, 5); writeln ('Bezeichnung  :');
9    Gotoxy (5, 7); writeln ('Lagermenge   :');
10   Gotoxy (5, 9); writeln ('Einzelpreis  :');
11   Gotoxy (5,11); writeln ('Zu-/Abgang   :');
12 end;


13 begin                                (* Buchen *)
14   B_Bildschirm;
15   repeat
16     Gotoxy (21,3); ClrEol; readln (B_ArtNr);
17     Suchen_Datensatz (B_LagDatei, B_Artikel,
                         B_ArtNr, B_SatzNr, B_Gefunden);
18     if B_Gefunden
19       then begin
20         with B_Artikel do
21         begin
22           Gotoxy (21,5); ClrEol;writeln (Bezeichnung);
23           Gotoxy (21,7); ClrEol;writeln (Menge);
24           Gotoxy (21,9); ClrEol;writeln (Preis:6:2);
25           Gotoxy (21,11); ClrEol; readln (MengenAenderung);
26           Menge := Menge + MengenAenderung;
27           Gotoxy (21,7); ClrEol;writeln (Menge);
28         end; (* with*)
29         seek (B_LagDatei, B_SatzNr - 1);
30         write (B_LagDatei, B_Artikel);
31       end
32       else begin
34         Gotoxy (10,20);
35         write ('Artikelnummer nicht vorhanden <RETURN>');
36         Weiter := ReadKey; (* 4 *)
       (* read (kbd, Weiter); * 3 *)
37         DelLine;
38       end;
39     Gotoxy (5,13); write ('Weiter (j/n) ');
40     Weiter := ReadKey; (* 4 *)
```

```
    (* read (kbd, Weiter); * 3 *)
41    DelLine;
42  until Weiter in ('n','N');
43  end;
```

Die Prozedur Buchen wird aus dem Hauptprogramm aufgerufen durch

```
Buchen (LagerDatei, Artikel);
```

Erläuterungen

Die Angaben, ob der Artikelsatz mit der angegebenen Artikelnummer in der Datei enthalten ist und unter welcher Satznummer er darin steht, wird in der Prozedur Suchen_Datensatz bereitgestellt. Da in dem Prozeduraufruf

```
17  Suchen_Datensatz (B_LagDatei, B_Artikel,
                      B_ArtNr, B_SatzNr, B_Gefunden);
```

bis auf B_ArtNr alle Parameter als Variablenparameter an die Prozedur Suchen_Datensatz übergeben werden, stehen anschließend folgende Angaben in der Prozedur Buchen zur Verfügung:

B_Artikel: die Daten des Artikels, die gemäß der in B_ArtNr eingegebenen Nummer in der Datei gelesen wurden,

B_SatzNr: die Satzadresse des aktuellen Datensatzes,

B_Gefunden: der Wert true oder false in Abhängigkeit davon, ob der Artikel in der Datei vorhanden ist.

In B_SatzNr steht also die Satznummer des gerade gelesenen Datensatzes, so daß mittels

```
29  seek (B_LagDatei, B_SatzNr - 1);
```

der Satzzeiger auf den Platz in der Datei positioniert wird, auf dem der aktuelle Artikelsatz gefunden wurde. Mit der anschließenden WRITE-Anweisung wird der Datensatz auf seine ursprüngliche Position zurückgeschrieben.

Das vor Dateioperationen erforderliche Öffnen der Datei brauchte nicht in der Prozedur Buchen vorgenommen zu werden, da diese Operation in der Prozedur Suchen_Datensatz bereits erfolgt ist.

5.3.5 Sortieren von Datensätzen

In vielen Anwendungsfällen ist es nützlich oder gar notwendig, daß die Datei nach einem bestimmten Ordnungsgesichtspunkt sortiert ist. Beim Ausdruck der Lagerliste beispielsweise wird häufig gefordert, daß sie nach aufsteigender Artikelnummer sortiert ist. Bei langen Listen ist sonst das Herausfinden einer bestimmten Artikelzeile sehr zeitaufwendig.
Damit eine solche Lagerliste nach Artikelnummern geordnet ausgegeben werden kann, muß die Lagerdatei vorher sortiert worden sein. Es gibt eine Vielzahl von Sortieralgorithmen. Wir wollen hier einen relativ "einfachen" Algorithmus anwenden.
Die Grundidee dieses Sortieralgorithmus besteht darin, daß die Datei zunächst nach dem Datensatz mit der niedrigsten Nummer durchsucht und dieser Satz auf Platz Null geschrieben wird. Dann wird die Datei ab Platz 1 nach dem Datensatz mit der zweitniedrigsten Nummer durchsucht. Dieser Satz wird auf Platz 1 gespeichert. In dieser Weise wird fortgefahren, bis die Datei geordnet ist.
Das Suchen des Datensatzes mit der jeweils niedrigsten Nummer erfordert folgende Schritte:

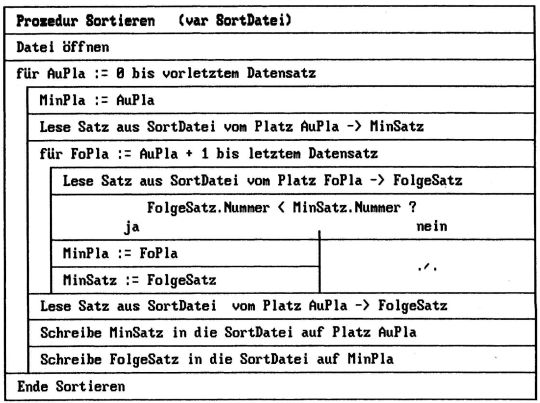
1. Der auf Platz Null in der Datei stehende Satz wird in die Variable MinSatz eingelesen. In die Variable MinPla wird die Platznummer des Satzes eingetragen.
2. Der auf dem nächsten Platz stehende Satz wird in die Variable FolgeSatz eingelesen. Seine Platznummer steht in der Variablen FoPla.
3. Die Artikelnummern der beiden Sätze werden miteinander verglichen. Ist die Artikelnummer in FolgeSatz kleiner als die im MinSatz, wird der Inhalt von FolgeSatz in die Variable MinSatz geschrieben. Der Variablen MinPlatz wird der Wert von FoPla zugewiesen.

```
wenn Artikelnummer.FolgeSatz < Artikelnummer.MinSatz
dann MinSatz := FolgeSatz
     MinPla  := FoPla
```

Ist die Artikelnummer in FolgeSatz nicht kleiner als die in MinSatz, so ist die Reihenfolge dieser beiden Datensätze richtig. Es wird keine Umspeicherung vorgenommen.

In MinSatz steht der Satz mit der kleinsten Artikelnummer. In MinPla ist seine Platznummer vermerkt. Nun wird der zweite Schritt erneut durchgeführt und der nächste Datensatz in FolgeSatz eingelesen. Wiederum werden der Vergleich und die eventuelle Umspeicherung wie im dritten Schritt vorgenommen. Diese beiden Schritte werden nun so lange durchgeführt, bis der letzte Satz der Datei nach FolgeSatz eingelesen und der Vergleich zwischen MinSatz und FolgeSatz erfolgt ist.
Auf Grund der Vergleiche zwischen MinSatz und FolgeSatz und der entsprechend vorgenommenen Umspeicherungen steht nun der Satz mit der kleinsten Artikelnummer in MinSatz. In MinPla ist seine Platznummer vermerkt.
Es erfolgt nun ein Platztausch. Weist beispielsweise MinPla die Platznummer 18 auf, so wird der Datensatz vom Platz Null auf den Platz 18 geschrieben. Der Inhalt von MinSatz wird auf Platz Null gespeichert. Damit steht der Satz mit der kleinsten Artikelnummer auf Platz Null. Im folgenden Sortiergang ist die zweitkleinste Artikelnummer zu suchen und auf Platz 1 zu schreiben.
Der Sortiergang wird mit dem 1. Schritt fortgesetzt. Jetzt wird der Datensatz vom Platz 1 in die Variable MinSatz gelesen, und in MinPla wird seine Platznummer vermerkt. Die Schritte 2 und 3 werden entsprechend durchlaufen. Die Sortierung ist schließlich beendet, wenn der vorletzte Datensatz in MinSatz eingelesen und mit dem nächsten, dem letzten Datensatz verglichen worden ist.
Der Sortierlauf wird über zwei geschachtelte Schleifen gesteuert. In der äußeren Schleife wird sichergestellt, daß jeder Platz mit Ausnahme des letzten im Schritt 1 angesprochen wird. In der inneren Schleife sind die jeweiligen Folgesätze einzulesen und zu vergleichen sowie die notwendigen Umspeicherungen vorzunehmen. Nach Beendigung der inneren Schleife erfolgt dann der Platztausch in der Datei.

Algorithmus
Diese Überlegungen zum Sortieren von Datensätzen führen zu folgendem Algorithmus:



Programm

```
1  procedure Sortieren_Datei (var SortDatei : LagerFile);
2  var MinPla, FoPla, AuPla  : integer;
       MinSatz, FolgeSatz    : ArtSatz;
3  begin
4    DateiOeffnen(SortDatei);
5    for AuPla := 0 to filesize(SortDatei) - 2 do
6    begin
7      MinPla := AuPla;
8      seek (SortDatei,AuPla); read (SortDatei,MinSatz);
9      for FoPla := AuPla+1 to filesize (SortDatei) - 1 do
10     begin
11       seek (SortDatei, FoPla);
12       read (SortDatei, FolgeSatz);
13       if FolgeSatz.Nummer < MinSatz.Nummer
14         then begin
15           MinPla := FoPla;
16           MinSatz := FolgeSatz;
17         end;
18     end;
19     seek (SortDatei,AuPla); read (SortDatei, FolgeSatz);
20     seek (SortDatei,MinPla); write (SortDatei,FolgeSatz);
21     seek (SortDatei,AuPla); write (SortDatei, MinSatz);
22   end;
23 end;
```

Die Prozedur wird aus dem Hauptprogramm aufgerufen durch

```
Sortieren_Datei (LagerDatei);
```

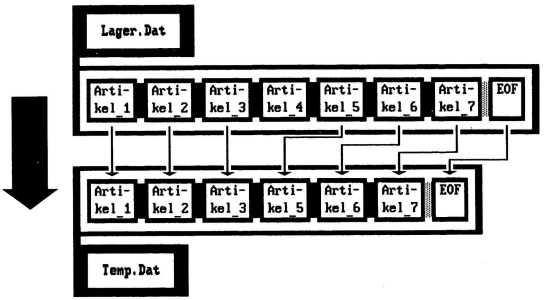
Erläuterungen
In den beiden Schleifen wurde für die Angabe der Austrittsbedingungen die Funktion FILESIZE verwendet. Da FILESIZE die Satznummer angibt, die um 1 höher ist als die Platznummer, wird durch

```
filesize (SortDatei) -2
```

die Nummer des vorletzten Platzes der Datei bestimmt. Entsprechend wird durch filesize (SortDatei) – 1 die Nummer des letzten Platzes ermittelt.
In der Prozedur wird zwar die Datei-Öffnungsroutine angesprochen, aber keine Anweisung zum Schließen der Datei gegeben. Die CLOSE-Anweisung erfolgt erst am Ende des Hauptprogramms. In der Version 4.0 von Turbo-Pascal wird nämlich im Gegensatz zur Version 3.x das Programm abgebrochen, wenn eine Close-Anweisung auf eine nicht geöffnete Datei angewendet wird.

5.3.6 Löschen von Datensätzen

Zum Umfang eines Lagerverwaltungsprogramms gehört auch eine Prozedur, mit der Artikelsätze aus der Lagerdatei gelöscht werden können. Die Prozedur ist so anzulegen, daß der Sachbearbeiter die Nummer des Artikels eingibt, der aus der Datei "gestrichen" werden soll.
Einen speziellen Befehl zum Löschen eines Datensatzes gibt es nicht in Turbo-Pascal. Das Löschen muß deshalb über eine Reihe von Programmschritten bewerkstelligt werden. Die Programmidee besteht darin, daß alle Sätze bis auf den zu löschenden in eine Zwischendatei kopiert werden.



Anschließend wird die Ursprungsdatei gelöscht und die Zwischendatei auf den Namen der Ursprungsdatei umbenannt. Der Algorithmus wird dann wie folgt aufgebaut:

Algorithmus

```
Prozedur Loeschen (var L_LagDatei; var L_Artikel)
  Bildschirmmaske
  Eingabe LoeNummer
  Suchen_Datensatz (L_Datei, _Artikel, L_SatzNr,L_Gefunden
                    gefunden ?
        ja                                  nein
  Artikelsatz anzeigen                  Fehlermeldung:
                                        "Artikelnummer nicht
                                         vorhanden"
              Löschen  ?
    ja                  nein
  Kopieren              ./.
  Eingabe Weiter (j/n)
Bis Weiter = 'n' oder 'N'
Ende Prozedur Loeschen
```

Prozedur Kopieren	
Einrichten einer Zwischendatei	
Satzzeiger auf Platz Null von L_LagDatei stellen	
NrDatenSatz := 1	
Solange nicht EOF (L_LagDatei)	
Lesen Artikelsatz aus L_LagDatei	
NrDatenSatz <> L_SatzNr ?	
ja	nein
Schreibe Artikelsatz in Zwischendatei	./.
Erhöhen NrDatenSatz um 1	
Schliepen Zwischendatei und L_LagDatei	
Löschen L_LagDatei	
Zwischendatei in L_LagDatei umbenennen	
Ende Prozedur Kopieren	

Programm

```
1  procedure Loeschen (var L_LagDatei : LagerFile;
                       var L_Artikel  : ArtSatz);
2  var LoeNummer           : integer;
       NrDatSatz, L_SatzNr : longint; (* 4 *)
                           (* integer;  * 3 *)
       ZwiDatei            : LagerFile;
       L_Gefunden          : boolean;
       Richtig, Weiter     : char;


3  procedure Kopieren;
4  begin
5    assign (ZwiDatei,'Temp.Dat');
6    rewrite (ZwiDatei);
7    seek (L_LagDatei,0);
8    NrDatSatz := 1;
9    while not eof (L_LagDatei) do
```

```
10    begin
11      read (L_LagDatei, L_Artikel);
12      if NrDatSatz <> L_SatzNr
          then write (ZwiDatei, L_Artikel);
13      NrDatSatz := NrDatSatz+1;
14    end;
15    close (ZwiDatei);
16    close (L_LagDatei);
17    erase (L_LagDatei);
18    rename (ZwiDatei,'Lager.Dat');
19  end;


20  begin                                  (*Loeschen*)
21    DateiOeffnen (L_LagDatei);
22    Bildschirmmaske;
23    Gotoxy (5,1); ClrEol;
24    write ('LAGERVERWALTUNG: ARTIKEL LOESCHEN');
25    repeat
26      Gotoxy (21,3); ClrEol; readln (LoeNummer);
27      Suchen_Datensatz (L_LagDatei, L_Artikel,
28                        LoeNummer,L_SatzNr,L_Gefunden);
29      if L_Gefunden
30        then begin
31          with L_Artikel do
32          begin
33            Gotoxy (21,5); ClrEol;writeln (Bezeichnung);
34            Gotoxy (21,7); ClrEol;writeln (Menge);
35            Gotoxy (21,9); ClrEol;writeln (Preis:6:2);
36            Gotoxy (5,13); write ('Loeschen (j/n)');
37            Richtig := ReadKey; (* 4 *)
38          (* read (kbd, Richtig); * 3 *)
39            DelLine;
40          end; (*with*)
41          if Richtig in ['j','J']
42            then Kopieren;
43        end
44        else begin
45          Gotoxy (10,20);
46          write ('Artikelnummer nicht vorhanden <RETURN>');
47          Weiter := ReadKey; (* 4 *)
48        (* read (kbd, Weiter); * 3 *)
49          DelLine;
50        end;
51      Gotoxy (5,13); write ('weiter ? (j/n)');
52      Weiter := ReadKey; (* 4 *)
53    (* read (kbd, Weiter); * 3 *)
54      DelLine;
55    until Weiter in ['n','N'];
56  end;
```

Die Prozedur wird aus dem Hauptprogramm aufgerufen durch

```
Loeschen (LagerDatei, Artikel);
```

Erläuterungen

In der Prozedur werden die Daten des Artikels, der gelöscht werden soll, zunächst angezeigt, damit der Benutzer noch einmal entscheiden kann, ob er den Satz tatsächlich löschen will. Nach der Bestätigung wird die Prozedur Kopieren aufgerufen. In ihr werden alle Datensätze bis auf den zu löschenden in die Zwischendatei eingetragen (Zeilen 11 bis 13). Ob ein Artikelsatz in die Zwischendatei eingetragen wird, hängt vom Ausgang des Vergleichs der beiden Satznummern ab. Der Eintrag in die Zwischendatei erfolgt nur dann, wenn die Nummer des zu löschenden Satzes ungleich der Satznummer des gerade gelesenen Satzes ist.
Ist das Ende der Lagerdatei erreicht, so werden die beiden offenen Dateien in den Zeilen 15 und 16 geschlossen. Mittels

```
17   erase (L_LagDatei);
```

wird die ursprüngliche Lagerdatei auf dem externen Speicher gelöscht (erase = engl.: auslöschen). Anschließend wird durch

```
18   rename (ZwiDatei, 'Lager.Dat');
```

die Zwischendatei auf den physischen Namen Lager.Dat umbenannt (rename = engl.: umbenennen).
Allgemein wird mit der Prozedur

erase (Dateivariable)

die Datei, die über den Assign-Befehl mit der Dateivariablen verbunden ist, auf dem externen Speicher gelöscht. Eine Datei darf nur dann gelöscht werden, wenn sie geschlossen ist.
Der formale Aufbau der RENAME-Prozedur entspricht dem der Assign-Prozedur:

rename (Dateivariable, 'Physischer_Dateiname')

Auch die RENAME-Prozedur darf nur auf geschlossene Dateien angewendet werden.
Das Löschen von Datensätzen aus einer Datei ist recht umständlich. Es gibt aber keinen anderen Weg, eine bestehende Datei zu kürzen.

Deshalb wird in Programmsystemen, die recht umfangreiche Dateien verwalten, ein anderes Verfahren angewandt. Dort werden die Datensätze, die zu löschen sind, mit einem besonderen Kennzeichen versehen. Beispielsweise könnten im Falle der Lagerdatei die zu löschenden Artikelsätze mit einer negativen Artikelnummer versehen werden. Von Zeit zu Zeit wird dann in einem Programmlauf eine neue Datei aufgebaut, in die die Sätze mit negativen Artikelnummern nicht übernommen werden.

5.3.7 Zusammenstellung des Programms Lagerverwaltung

Das Programm ist so aufgebaut, daß die einzelnen Funktionen des Lagerverwaltungsprogramms stets aus einem Menu angewählt und aus dem Hauptprogramm aufgerufen werden. Nach Beendigung der Funktion kehrt das Programm in das Menu zurück. Der Benutzer kann nun die nächste Funktion auswählen.

```
program Lagerverwaltung1;
uses CRT;              (* 4 *)
type ArtSatz     = record
                     Nummer      : integer;
                     Bezeichnung : string (.30.);
                     Menge       : integer;
                     Preis       : real;
                   end;
     LagerFile   = file of ArtSatz;
var  Artikel     : ArtSatz;
     LagerDatei  : LagerFile;
     Wahl        : char;
     Ende        : boolean;


procedure DateiOeffnen (var LagDatei: LagerFile);

procedure Bildschirmmaske;

procedure Suchen_Datensatz     (var S_LagDatei : LagerFile ;
                                var S_Art      : ArtSatz;
                                    S_ArtNr    : integer;
                                var S_SatzNr   : Longint; (* 4 *)
                                                 (* integer;  * 3 *)
                                var S_Gefunden : boolean);
```

```
procedure Buchen                (var B_LagDatei : LagerFile;
                                    B_Artikel   : ArtSatz);

procedure Eingabe_Artikel       (var E_LagDatei : LagerFile;
                                 var E_Artikel  : ArtSatz);

procedure Datei_Lesen_Anzeigen (var D_LagDatei : LagerFile;
                                 var D_Artikel  : ArtSatz);

procedure Sortieren_Datei       (var SortDatei  : LagerFile);

procedure Loeschen              (var L_LagDatei : LagerFile;
                                 var L_Artikel  : ArtSatz);


procedure Auswahl (var Menu : char);
begin
  ClrScr;
  Gotoxy ( 5, 5); writeln ('L A G E R V E R W A L T U N G');
  Gotoxy ( 5, 6); writeln ('----------------------------');
  Gotoxy ( 5, 8); writeln ('Eingabe Artikeldaten .....: 1');
  Gotoxy ( 5,10); writeln ('Artikeldaten anzeigen ....: 2');
  Gotoxy ( 5,12); writeln ('Mengenaenderungen buchen .: 3');
  Gotoxy ( 5,14); writeln ('Sortieren ................: 4');
  Gotoxy ( 5,16); writeln ('Artikel loeschen .........: 5');
  Gotoxy ( 5,20); writeln ('Ende .....................: 9');
  Gotoxy (10,22); write ('Auswahl ..:');
  Menu := ReadKey;  (* 4 *)
(* read (kbd, Menu); * 3 *)
end;


begin                                   (* HAUPTPROGRAMM *)
  assign (LagerDatei,'Lager.Dat');
  DateiOeffnen (LagerDatei);
  repeat
    Ende := false;
    Auswahl (Wahl);
    case Wahl of
      '1' : Eingabe_Artikel (LagerDatei, Artikel);
      '2' : Datei_Lesen_Anzeigen (LagerDatei, Artikel);
      '3' : Buchen (LagerDatei, Artikel);
      '4' : Sortieren_Datei (LagerDatei);
      '5' : Loeschen (LagerDatei, Artikel);
      else  Ende := true;
    end; (*case*)
  until Ende;
  close (LagerDatei);
end.
```

5.3.8 Zusammenfassung

- Mittels **(*$I−*)** wird der **automatische Programmabbruch** nach fehlerhaften Eingabe-/Ausgabeoperationen **ausgeschaltet**. Mit **(*$I+*)** wird er wieder **aktiviert**. Bei ausgeschaltetem automatischen Abbruch kann der vom System gemeldete Fehler über **IOResult** abgefragt werden.
- Mittels **eof** (Dateivariable) kann auf das Ende der Datei abgefragt werden.
- Die Funktion

  filesize (Dateivariable)

  liefert die Anzahl der Datensätze, die in der Datei eingetragen sind. Die Zählweise beginnt bei 1.
- Die Prozedur **seek** (Dateivariable, Platznummer) stellt den Satzzeiger auf den durch Platznummer angegebenen Datensatz. Die Zählweise für die Platznummern beginnt mit Null.
- Nach Durchführung einer Lese- bzw. Schreiboperation steht der Satzzeiger auf dem Beginn des nächsten Datensatzes.
- Mittels der Funktion **filepos** (Dateivariable) wird festgestellt, auf dem wievielten Datensatz der Satzzeiger steht.
- Zum Löschen eines Datensatzes müssen alle Datensätze einer Datei bis auf den zu löschenden in eine zweite Datei kopiert werden.
- Die Prozedur

  erase (Dateivariable)

  löscht die mit der Dateivariablen verbundene physische Datei auf dem externen Datenträger.
- Die Prozedur

  rename (Dateivariable, Name_physische_Datei)

  benennt die der Dateivariablen zugeordnete physische Datei um. Die physische Datei erhält den Namen, der durch eine String-Konstante oder in einer String-Variablen angegeben wird.

5.3.9 Übungen

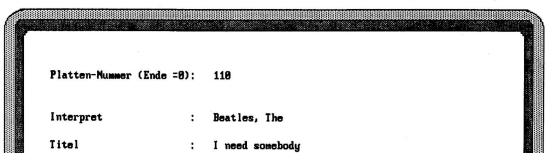
1. Aufgabe
In den obigen Ausführungen ist die Prozedur Datei_Lesen_Anzeigen nicht dargelegt worden. Schreiben Sie die Prozedur! Beachten Sie dabei den im Programm Lagerverwaltung1 formulierten Prozedurkopf. Weitere Verarbeitungshinweise können Sie dem Programm Lagerdatei_Aufbau_Anzeigen (s. Kap. 5.2.1) entnehmen.

2. Aufgabe
Entwickeln Sie eine Prozedur, mit der der Wert aller im Lager befindlichen Artikel ermittelt und auf dem Bildschirm angezeigt wird.

3. Aufgabe
Um eine bessere Übersicht über seine Schallplattensammlung zu erhalten, möchte ein Musikliebhaber seine Schallplatten in einer Datei erfassen. Zu jeder Platte gibt er ein:

Nummer
Interpret
Titel

Wie aus der Bildschirmabbildung zu entnehmen ist, wird die Eingabe so lange vorgenommen, bis als Nummer eine Null eingetippt wird. Anschließend ist der Inhalt der Plattendatei am Bildschirm anzuzeigen.



Stellen Sie sicher, daß neue Datensätze stets am Ende der Datei eingefügt werden!

4. Aufgabe

Um Doppelerfassungen zu vermeiden, ist die Plattendatei nach Eingabe der Plattennummer darauf zu überprüfen, ob die Schallplatte bereits in der Datei erfaßt ist.
Entwickeln Sie zu Ihrem Programm in Aufgabe 3 eine entsprechende Suchfunktion oder Suchprozedur!

5.3.10 Lösungen

1. Aufgabe

```
procedure Datei_Lesen_Anzeigen (var L_LagDatei : LagerFile;
                                var L_Artikel  : ArtSatz);
var Weiter : char;
begin
  DateiOeffnen(L_LagDatei);
  ClrScr;
  while not eof (L_LagDatei) do
  begin
    read (L_LagDatei, L_Artikel);
    with L_Artikel do
      writeln (Nummer:6,' ',Bezeichnung,
               ' ':32-length (Bezeichnung), Menge:10, Preis:10:2);
  end;
  writeln; writeln ('Weiter <RETURN> ');
  Weiter := ReadKey;  (* 4 *)
(* read (kbd, Weiter); * 3 *)
end;
```

2. Aufgabe

```
procedure AufSummierung (var A_LagDatei : LagerFile;
                         var A_Artikel  : ArtSatz);
var Weiter : char;
    Summe  : real;
begin
  DateiOeffnen(A_LagDatei);
  ClrScr;
  Summe := 0;
  while not eof (A_LagDatei) do
  begin
    read (A_LagDatei, A_Artikel);
    with A_Artikel do
      Summe := Summe + Menge * Preis;
  end;
  writeln ('Lagerwert',Summe:10:2);
  writeln; writeln ('Weiter <RETURN> ');
  Weiter := ReadKey;  (* 4 *)
(* read (kbd, Weiter); * 3 *)
end;
```

3. Aufgabe

```
program Schallplattenarchiv;
uses CRT;           (* 4 *)
type DiskSatz     = record
                      Nummer    : integer;
                      Interpret : string (.20.);
                      Titel     : string (.30.);
                    end;
var  Platte       : DiskSatz;
     PlattenDatei : file of DiskSatz;


procedure Bildschirmmaske;
begin
  ClrScr;
  Gotoxy (5,3); writeln ('Platten-Nummer (Ende =0):');
  Gotoxy (5,5); writeln ('Interpret               :');
  Gotoxy (5,7); writeln ('Titel                   :');
end;
```

```
procedure DateiOeffnen;
begin
  (*$I-*)
  reset (PlattenDatei);
  if IORESULT <> 0
    then rewrite (Plattendatei);
  if IORESULT <> 0
    then begin
      writeln ('Datei kann nicht geöffnet werden!');
      halt;
  (*$I+*)
end;

procedure Eingabe;
begin
  Bildschirmmaske;
  DateiOeffnen;
  Gotoxy (32,3); ClrEol; readln (Platte.Nummer);
  while  Platte.Nummer <> 0 do
  begin
    Gotoxy (32,5); ClrEol; readln (Platte.Interpret);
    Gotoxy (32,7); ClrEol; readln (Platte.Titel);
    seek (Plattendatei, filesize (Plattendatei));
    write (Plattendatei,Platte);
    Gotoxy (32,3); ClrEol; readln (Platte.Nummer);
  end;
  close (Plattendatei);
end;


procedure Datei_Lesen_Anzeigen;
begin
  ClrScr;
  DateiOeffnen;
  while not eof (Plattendatei) do
  begin
    read (Plattendatei, Platte);
    with Platte do
      writeln (Nummer:6,' ',Interpret,' ':22-length (Interpret),
               Titel);
  end;
  close (PlattenDatei);
end;


begin                                    (* HAUPTPROGRAMM *)
  assign (PlattenDatei,'Platten.Dat');
  Eingabe;
  Datei_Lesen_Anzeigen;
end.
```

4. Aufgabe

```
function Gefunden (SuchNummer : integer): boolean;
var GibtEs : boolean;
begin
  GibtEs := false;
  while  not eof (PlattenDatei) and( not GibtEs) do
  begin
    read (Plattendatei, Platte);
    if Platte.Nummer = SuchNummer
      then GibtEs := true;
  end;
  Gefunden := GibtEs;
end;
```

6 INDEX-SEQUENTIELLE DATEI

6.1 Kennzeichen einer index-sequentiellen Datei

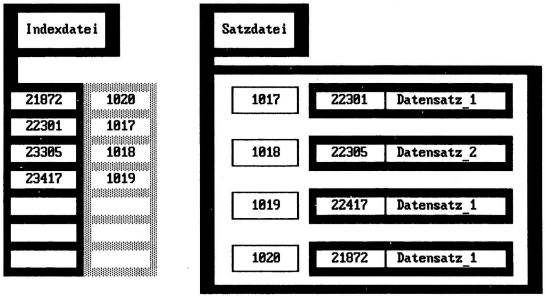
Einführungsbeispiel: Das Sortiment eines Großhandelsunternehmens besteht aus 9000 Artikeln. Die Lagerbuchhaltung wird mittels Computer durchgeführt. Die Artikeldaten sind in einer sequentiellen Datei gespeichert. Die Mitarbeiter beklagen sich, daß es beim Buchen der Lagerzu- und -abgänge manchmal sehr lange dauert, bis nach der Eingabe der Artikelnummer der Artikelsatz angezeigt wird und sie die Buchung vornehmen können.

6.1.1 Sequentielle Datei – index-sequentielles Dateisystem

Der Grund für die beklagte geringe Verarbeitungsgeschwindigkeit liegt darin, daß die Lagerdatei sequentiell organisiert ist. Soll beispielsweise ein Artikelsatz aus der Datei in den Arbeitsspeicher eingelesen werden, der an 8000. Stelle in der Datei steht, so sind unter Umständen 8000 Lese- und Vergleichsoperationen durchzuführen, ehe der gewünschte Datensatz in den Arbeitsspeicher eingelesen wird.
Die Verarbeitungsgeschwindigkeit kann jedoch erhöht werden, wenn die Lagerdatei **index-sequentiell** organisiert wird. Die schnellere Verarbeitung bei index-sequentieller Dateiorganisation wird dadurch erreicht, daß zwei Dateien geführt werden. Die eine Datei, auch **Satzdatei** genannt, beinhaltet die Datensätze. Bei einer Lagerdatei beispielsweise kann der Datensatz aus den Angaben zu

Artikelnummer
Artikelbezeichnung
Preis
Menge

bestehen. In der zweiten Datei, die **Indexdatei** genannt wird, werden die Schlüssel, z. B. die Artikelnummer, und die Satzadresse gespeichert. Die Satzadresse gibt an, wo der zum Schlüssel zugehörige Datensatz in der Satzdatei gespeichert ist. Die Indexdatei ist also vergleichbar mit dem Sachwortverzeichnis eines Buches. In einem solchen Verzeichnis wird zu den Stichwörtern jeweils eine Seitenzahl angegeben, auf der nähere Erläuterungen zu finden sind.
Unter einem Schlüssel versteht man allgemein eine Angabe, die eine Person oder eine Sache genau kennzeichnet. Ein Artikel wird beispielsweise durch eine Artikelnummer genau bestimmt. Deshalb eignet sich die Artikelnummer gut als Schlüssel zur Kennzeichnung eines Artikelsatzes in einer Lagerdatei.
Mit Hilfe des Schlüssels kann man "direkt" auf die Satzdatei zugreifen. Das sei am Beispiel des Lesens eines Datensatzes dargelegt.



6.1.2 Lesen einer index-sequentiellen Datei

Das Lesen eines Datensatzes wird an Hand der obigen Abbildung erläutert. Als Schlüsselbegriff dient hier die Artikelnummer. Es soll der Artikel mit der Artikelnummer 23417 gelesen werden. Zunächst

durchsucht das System die Indexdatei nach dem Schlüssel 23417. Hinter diesem Schlüssel ist die Satzadresse 1019 vermerkt. Anschließend greift das System auf den Satz mit der Adresse 1019 zu und liest den Datensatz ein. Mit nur einem Lesezugriff wird der Datensatz aus der Satzdatei in den Arbeitsspeicher eingelesen.
In allgemeiner Form kann man den Lesevorgang wie folgt beschreiben:

1. Eingabe des Suchbegriffs (Schlüssel).
2. Suchen des Schlüssels in der Indexdatei.
3. Lesen des Datensatzes in der Satzdatei unter der Adresse, die in der Indexdatei angegeben ist.

6.1.3 Schreiben in eine index-sequentielle Datei

Soll ein weiterer Datensatz eingetragen werden, so wird der Datensatz an die bestehende Satzdatei angefügt. Dabei merkt sich das System die Satzadresse. Die Satzadresse gibt an, unter welcher laufenden Satznummer der Datensatz in die Satzdatei geschrieben wurde.
Anschließend trägt das System den Schlüssel, beispielsweise die Artikelnummer, sowie die Satzadresse in die Indexdatei ein. Die Indexdatei-Einträge werden entsprechend dem Schlüsselwert in aufsteigender Reihenfolge sortiert angeordnet.
Das Speichern eines Datensatzes in einer index-sequentiell organisierten Datei erfolgt allgemein in drei Schritten:

1. Eingabe des Datensatzes.
2. Anfügen des Datensatzes an die Satzdatei.
3. Eintragen des Schlüsselbegriffes und der Satzadresse in die Indexdatei.

6.1.4 Zusammenfassung

- Ein index-sequentielles Dateisystem besteht aus einer Satzdatei und mindestens einer Indexdatei.
- Die Satzdatei enthält die Datensätze.
- Die Indexdatei enthält die Schlüssel und die Adreßverweise.
- Ein Schlüssel kennzeichnet einen Datensatz.

6.1.5 Übungen

1. Aufgabe
Welche Daten werden in die Indexdatei eingetragen?

2. Aufgabe
Nennen Sie einen geeigneten Schlüssel für die Kraftfahrzeugdatei eines Transportunternehmens!

3. Aufgabe
In welcher Reihenfolge werden die Daten- und Indexeinträge vorgenommen?

4. Aufgabe
Wie erfolgt der Zugriff auf einen bestimmten Datensatz in einem index-sequentiellen Dateisystem?

6.1.6 Lösungen

1. Aufgabe
In die Indexdatei werden der Schlüssel und die Satzadresse zu einem bestimmten Datensatz eingetragen.

2. Aufgabe
Die Kfz-Nummer ist ein geeigneter Schlüssel.

3. Aufgabe
Zuerst wird der Datensatz in die Satzdatei eingetragen. Anschließend werden die Schlüssel und die Satzadresse des gerade gespeicherten Datensatzes in die Indexdatei geschrieben.

4. Aufgabe
Der Schlüssel, der den Datensatz kennzeichnet, wird in der Indexdatei gesucht, und die zugehörige Satzadresse wird festgestellt. Anschließend wird auf die Stelle der Satzdatei zugegriffen, die der Satzadresse entspricht.

6.2 Index-sequentielle Dateiorganisation unter Turbo-Pascal

Um unter Turbo-Pascal mit index-sequentiellen Dateien arbeiten zu können, benötigen Sie das **Turbo Access System** für die Version 3.x bzw. die **DATABASE TOOLBOX** für Version 4.0 von Turbo-Pascal. Diese Programmsysteme gehören nicht zum normalen Lieferumfang von Turbo-Pascal. Sie sind gesondert zu erwerben.
Die DATABASE TOOLBOX enthält u. a. den Funktionsumfang des Turbo Access Systems. Die nachstehenden Ausführungen beziehen sich somit auf beide Programmsysteme.

6.2.1 Installationshinweise

Turbo Access System
Wenn die Turbo-Pascal Vers. 3.0 einsetzen, so müssen Sie das Turbo Access System verwenden. Es besteht aus 4 Programm-Modulen:

ACCESS3.BOX
GETKEY.BOX
ADDKEY.BOX
DELKEY.BOX

Die ACCESS3.BOX befindet sich in der Regel bereits auf der Turbo-Pascal-Systemdiskette. Wenn Sie auf einer Diskettenanlage arbeiten, sollten Sie die übrigen drei *.BOX-Programme auf die Pascal-Systemdiskette kopieren.

DATABASE TOOLBOX
Bei Turbo-Pascal Vers. 4.0 ist DATABASE einzusetzen.
Die DATABASE-TOOLBOX wird mit folgenden Schritten installiert:

Installation bei Festplattenanlage: Wir gehen davon aus, daß Sie das Turbo-System in einem Verzeichnis mit dem Namen Turbo gespeichert haben. Richten Sie im Turbo-Verzeichnis ein weiteres Verzeichnis mit dem Name access ein, und wechseln Sie in das neue Verzeichnis:

md access <RETURN>
cd access <RETURN>

Legen Sie die DATABASE-Originaldiskette Nr. 1 in das Laufwerk a: und kopieren Sie mittels

copy a:\ta*.pas
copy a:\ta*.exe

die benötigten Programme in das Unterverzeichnis access!
Starten Sie das Programm **tainst** <RETURN>! Dadurch wird das ACCESS-System eingerichtet. Beim Installationslauf geben Sie auf die Frage nach

Turbo Pascal Directory: <RETURN>
Turbo Access Directory: **c:\turbo** <RETURN>

ein! Damit haben Sie dem System die erforderlichen Informationen mitgeteilt.
Tragen Sie das ACCESS-Unterverzeichnis in die AUTOEXEC.BAT-Datei hinter Path ein! Nähere Erläuterungen dazu können Sie dem rororo-Taschenbuch 8145 (Peter Freese, Standardbetriebssystem MS-DOS) entnehmen.

Installation bei Diskettenanlagen (360er Format): Legen Sie eine formatierte Leerdiskette in das Laufwerk b:! Stellen Sie das System mit b: <RETURN> auf das Laufwerk b: um!
Legen Sie die DATABASE-Originaldiskette Nr. 1 in das Laufwerk a: und kopieren Sie mittels

copy a:\ta*.pas
copy a:\ta*.exe

die benötigten Programme auf die Diskette im Laufwerk b:!
Schaffen Sie sich eine Turbo-Pascal-Systemdiskette, die folgende Files enthält, an:

- Command.COM
- Autoexec.BAT
- Turbo.EXE
- Turbo.TPL
- TPC.EXE
- Keybgr.exe

Das ist die Konfiguration, die bei einem 360er Laufwerk auf eine Dis-

kette paßt. Legen Sie diese Turbo-Pascal-Systemdiskette in das Laufwerk a:!
Starten Sie nun das Programm **tainst** <RETURN>! Dadurch wird das ACCESS-System eingerichtet.
Im Installationslauf werden Sie gefragt, in welchen Verzeichnissen und/oder Laufwerken sich der Turbo-Compiler und das Turbo-ACCESS-System befinden.

Turbo-Directory: **a:**
ACCESS-Directory: **b:**

Nach Abschluß des Installationslaufs tragen Sie in die AUTOEXEC.BAT-Datei ein:

Keybgr. exe
Path A:\B:\
Turbo

Das System ist nun in der Weise eingerichtet, daß Sie Ihre Anwendungsprogramme und Dateien auf dem Laufwerk b: führen können.

6.2.2 Steuerungsparameter

Der Aufbau und die Verwaltung einer index-sequentiellen Datei werden am Beispiel eines Lagerdatei-Verwaltungsprogramms dargestellt.

Festlegung von Datensatz und Schlüssel
Eine Lagerdatei setzt sich aus Artikelsätzen zusammen. Der Artikelsatz wird wie folgt festgelegt:

```
type str30    = string(.30.);
     str5     = string (.5.);
     LagerRec = RECORD
                  RecStatus     : LongInt; (* 4 *)
                                  (* integer;  * 3 *)
                  Nr            : str5;
                  Bezeichnung   : str30;
                  Menge         : integer;
                  Preis         : real;
                end;
```

In der Variablen RecStatus wird vermerkt, welchen Satzstatus der Artikelsatz hat. Der Satzstatus gibt an, ob der Datensatz gültig oder nicht mehr gültig ist.
Als Schlüsselbegriff wird hier die Artikelnummer gewählt. Das Programm soll also in der Weise eingerichtet werden, daß man über die Artikelnummer direkt auf den Artikelsatz zugreifen kann. Da in Turbo-Pascal der Schlüsselbegriff vom Typ **string** sein muß, wird die Artikelnummer als eine 5stellige Zeichenkette festgelegt. Die Länge eines Schlüsselbegriffes darf in Turbo-Pascal bis zu 255 Zeichen umfassen.
Für die Einrichtung der Index-Datei, die unter Turbo-Pascal in der Datenstruktur eines B-Baumes geführt wird, werden einige Angaben in Form von vordefinierten Konstanten benötigt. Die Datenstruktur des B-Baumes wird hier nicht näher erläutert, da ihr Verständnis keine unbedingte Voraussetzung für die Anwendung des Turbo-Access-Systems ist. Nähere Informationen zum B-Baum finden Sie in dem Lehrbuch von N. Wirth (Algorithmen und Datenstrukturen, Stuttgart 1979).

Festlegung der Steuerungsparameter für das Turbo-Access-System (*3*)

MaxDataRecSize: Unter MaxDataRecSize ist dem System mitzuteilen, wie viele Zeichen (Byte) der Datensatz umfaßt. Bezogen auf den obigen Artikelsatz ist anzugeben:

```
const MaxDataRecSize = 45;
```

Berechnungsbeispiel:	Speicherplatz in Byte	
Satzstatus: integer	2	
Artikelnummer: string (.5.)	6	(incl. Längenbyte)
Artikelbezeichnung: string (.30.)	31	(incl. Längenbyte)
Preis: real	4	
Menge: integer	2	
Summe	45	

MaxKeyLen: Mit MaxKeyLen wird die maximale Länge des Schlüsselbegriffes festgelegt. Wird in einem Programm mit mehreren Index-

dateien und somit auch mit mehreren Schlüsselbegriffen gearbeitet, so ist die Länge des längsten Schlüssels zu vereinbaren. In unserem Lagerbeispiel ist der Schlüssel 5 Zeichen lang.

```
const MaxKeyLen = 5;
```

Weiterhin sind einige Angaben zu machen, die das DATABASE-System für die Strukturierung des B-Baumes benötigt. Über diese Angaben kann der Programmierer die Struktur des Baumes beeinflussen. In der Regel kann man jedoch die vom Hersteller angegebenen Standardwerte übernehmen.

PageSize: Unter PageSize ist anzugeben, wie viele Schlüssel in einem Indexdateisatz eingetragen werden sollen. Die Anzahl muß eine gerade Zahl zwischen 2 und 254 sein. Als Standardwert wird hier 24 angenommen.

```
const PageSize = 24;
```

Order: Unter Order ist der Wert von PageSize/2 anzugeben. In unserem Beispiel beträgt somit der Wert für Order 12.

```
const Order = 12;
```

PageStackSize: Unter PageStackSize wird angegeben, wieviel Indexdateisätze sich gleichzeitig im Arbeitsspeicher befinden sollen. Als Mindestwert ist 3 anzugeben. Je größer dieser Wert angesetzt wird, desto höher wird die Verarbeitungsgeschwindigkeit, da zur Bearbeitung der Indexdatei kaum noch auf die Diskette zugegriffen werden muß. Hier wird der Wert 8 angenommen.

```
const PageStackSize = 8;
```

MaxHeight: Mit MaxHeight wird die maximale Höhe der Baumstruktur vereinbart. Sie errechnet sich aus der Formel

$$\text{MaxHeight} = \text{int}(\log(\text{Anzahl Einträge})/(\log(\text{PageSize}) \times 0{,}5)) + 3$$

Berechnungsbeispiel:
Soll die Anzahl der Schlüsseleinträge 65000, die Größe für PageSize 24 betragen, so errechnet sich MaxHeight aus

$$\text{int}(4{,}8129134/1{,}3802112 \times 0{,}5) + 3 \approx 5$$

MaxHeight wird festgelegt durch:

```
const MaxHeight = 5;
```

Diese oben dargestellten Werte sind zu Beginn des Programms anzugeben. Ferner müssen die Access-Module als sogenannte Include-Files aufgeführt werden. Ein **Include-File** ist eine vorformulierte Datei, die aus einem Programm heraus beim Compilieren aufgerufen und in das Programm eingebunden wird. Es wird zusammen mit dem Programm übersetzt.
Die einzuladende Datei wird durch (*$I Dateiname *) aufgerufen.
Der Programmkopf für das Lagerverwaltungsprogramm lautet dann:

```
program Lagerverwaltung2;
const MaxDataRecSize = 47;
      MaxKeyLen      =  5;
      PageSize       = 24;
      Order          = 12;
      PageStackSize  =  8;
      MaxHeight      =  5;
(*$I ACCESS3.BOX *)          (* Kein Semikolon! *)
(*$I GETKEV.BOX *)
(*$I ADDKEV.BOX *)
(*$I DELKEV.BOX *)
```

Von diesen 4 Include-Files muß ACCESS3.BOX als erste aufgeführt werden.
Mit diesen Angaben verfügt das Turbo-Pascal-System über die erforderlichen Informationen, um eine index-sequentielle Datei aufzubauen.

Festlegung der Steuerungsparameter für DATABASE (*4*)
Die DATABASE-Version 4.0 verfügt über ein Programm mit dem Namen TABUILD, das die erforderlichen Steuerparameter in einer Datei mit dem Namen TACCESS.TPU erzeugt. Für jede Anwendung, die index-sequentielle Dateien benutzt, wird ein eigenes TAC-

CESS.TPU-File geschaffen. Deshalb wird für sie ein eigenes Unterverzeichnis angelegt.
Nach dem Start des Turbo-Pascal-Systems rufen Sie zunächst mittels <Alt> <F> <O> die Betriebssystemebene auf und richten für das Lagerverwaltungssystem mit

md Lager <RETURN> (md = make directory)

ein Unterverzeichnis mit dem Name Lager ein. Mit exit <RETURN> kehren Sie in das Turbo-Pascal-System zurück.
Mit <Alt> <F> <C> rufen Sie das Kommando zum Wechseln des Unterverzeichnisses auf und teilen dem System in dem Eingabefenster den Namen des neuen Unterverzeichnisses **Lager** <RETURN> mit.
Anschließend aktivieren Sie mit <Alt> <E> den Editor und geben den Artikelsatz ein.

```
type str30    = string(.30.);
     str5     = string (.5.);
     LagerRec = RECORD
                  Recstatus     : LongInt;
                  Nr            : str5;
                  Bezeichnung   : str30;
                  Menge         : integer;
                  Preis         : real;
                end;

     MaxDataType = LagerRec;      (* Dies ist der größte RECORD
                                     im Dateisystem Lager *)
     MaxKeyType  = Str5;          (* Dies ist der längste Schluessel
                                     im Dateisystem Lager *)
```

In der Definition des Artikelsatzes LagerRec ist an erster Stelle eine Longint-Variable zu vereinbaren. Die Variable, hier mit Recstatus bezeichnet, wird vom System zur Kennzeichnung des Satzes benötigt, der entweder gültig oder nicht gültig sein kann. Ein gelöschter Datensatz wird vom System auf den Status nicht gültig gesetzt. Die Variable weist dann einen Wert ungleich Null auf.
Ferner ist unter MaxDataType die Länge des größten Datensatzes anzugeben. Da in dem Lagerbeispiel nur eine index-sequentielle Datei verwendet wird, wird dem System durch

MaxDataType = LagerRec;

mitgeteilt, daß LagerRec der längste Datensatz ist.

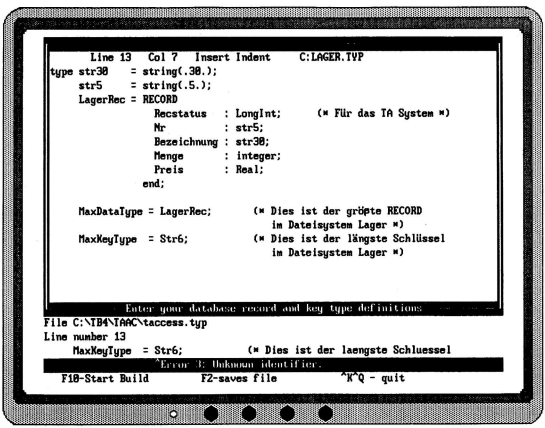
Schließlich ist der längste Schlüssel anzugeben. Da die Artikelnummer vom Typ str5 der einzige Schlüssel im Dateisystem Lager ist, wird die Schlüssellänge durch

MaxKeyType = str5;

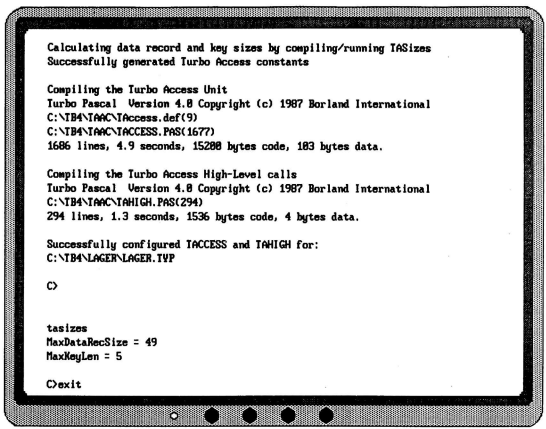
festgelegt.
Die Parameterangaben speichern Sie mit <F2> unter dem Namen **Lager.typ** ab. Anschließend verlassen Sie mit <Alt> <F> <O> das Turbo-Pascal-System zur Betriebssystemebene. Mit der Eingabe

tabuild Lager.typ <RETURN>

veranlassen Sie nun, daß die für das Dateisystem benötigten Dateien TACCESS.TPU und TAHIGH.TPU im Unterverzeichnis Lager aufgebaut werden.
Falls Tabuild einen Fehler in den Vereinbarungen feststellt, wird in einem Fenster die Vereinbarung angezeigt und unterhalb des Fensters der Fehler kommentiert:



Sie korrigieren den Fehler und speichern mit <F2> die Vereinbarung. Mit <F10> starten Sie das Programm Tabuild erneut. Nach erfolgreicher Compilation wird am Bildschirm folgende Meldung angezeigt:



Durch Eingabe von exit <RETURN> kehren Sie in das Turbo-Pascal-System zurück. Nach Betätigen von <Alt> <E> können Sie mit der Eingabe des Programms beginnen. Die ersten Zeilen des Programms:

```
program Lagerverwaltung2;
uses CRT, TACCESS;
(*$I Lager.typ*)
```

Hinter uses ist als weiteres Unit **TACCESS** anzugeben. Dadurch werden die Systemprogramme zur Verwaltung von index-sequentiellen Dateien aufgerufen. Ferner ist durch

```
(*$I Lager.typ*)
```

die Datei anzugeben, in der die Typdefinitionen gespeichert worden

sind. Diese Datei wird als Include-File beim Compilieren des Programms Lagerverwaltung in das Programm eingebunden. Die dort getroffenen Vereinbarungen brauchen im Programm Lagerverwaltung also nicht noch einmal eingegeben zu werden.

6.2.3 Zusammenfassung

- Zum Aufbau und zur Verwaltung einer index-sequentiellen Dateiorganisation wird für Turbo-Pascal Version 3.x das Turbo Access System und für die Version 4.0 die DATABASE TOOLBOX benötigt.
- Das Turbo-Access-System kann bis zu 65356 Datensätze, die DATABASE über 2 Mio Datensätze in einer Satzdatei verwalten.
- Vor Aufbau eines index-sequentiellen Dateisystems sind der Datensatzaufbau und der Schlüssel festzulegen.
- Der **Datensatz** ist vom **Typ RECORD**. Seine minimale Größe muß 8 Bytes, seine maximale Größe kann 64 KB betragen.
- Das erste Element des Datensatzes ist eine **Satzstatusvaribable** vom Typ integer (*3*) bzw. longint (*4*).
- Der **Schlüssel** ist vom **Typ STRING**. Seine Länge kann 1 bis 255 Bytes betragen.
- Im Turbo-Access-System müssen zu einigen Systemkonstanten Werte angegeben werden. Die Werte zu MaxDataRecSize und MaxKeyLen sind von der jeweiligen Aufgabenstellung abhängig. Bei den übrigen Systemkonstanten können die Standardvorgaben als Werte verwendet werden. (*3*)
- Im Turbo-Access-System sind die Programmodule im Quellprogramm als Include-Files anzugeben. Dabei muß (*$I ACCESS-3.BOX*) als erstes genannt werden. (*3*)
- Bei DATABASE wird für jede Anwendung ein eigenes Unterverzeichnis eingerichtet. (*4*)
- Der Datensatz sowie die Angaben zu MaxDataType und MaxKeyType werden in einer eigenen Typdatei erfaßt, die als Include-File im Quellprogramm anzugeben ist (*4*)
- Die Steuerparameter werden durch das Programm **TABUILD** erzeugt. (*4*)
  Im Quellprogramm ist als weiterer Unit **TACCESS** anzugeben. (*4*)

6.2.4 Übungen

1. Aufgabe
Eine Kfz-Datei soll aufgebaut werden. Der Datensatz besteht aus folgenden Elementen:

KfzNr	:	string (.8.)
Fabrikat	:	string (.20.)
Type	:	string (.15.)
FahrgestellNr	:	string (.17.)
Anschaffungsjahr	:	integer

a) Ergänzen Sie das fehlende, für das System wichtige Datenelement!
b) Im Turbo-Access-System ist die Länge des Datensatzes anzugeben. Bestimmen Sie die Länge in Byte!
c) Wie wird für das Turbo-Access-System der Wert für MaxKeyLen festgelegt, wenn die KfzNr als Schlüssel verwendet werden soll?

2. Aufgabe
Wie wird ein Include-File gekennzeichnet?

3. Aufgabe
Welchen Angaben müssen bei Verwendung von DATABASE in der Typdatei gemacht werden?

4. Aufgabe
Erstellen Sie eine Typdatei für den Datensatz von Aufgabe 1!

5. Aufgabe
Welche Information wird in der Satzstatusvaribalen vermerkt?

6. Aufgabe
Wie wird bei Turbo-Pascal 4.0 in einem Programm angegeben, daß mit DATABASE gearbeitet werden soll?

6.2.5 Lösungen

1. Aufgabe

a) Es fehlt die Satzstatusvariable, die als erstes Element des RECORDS anzugeben ist.
(*3*) Satzstatus : integer
(*4*) Satzstatus: longint

b) Der Datensatz besteht aus 68 Bytes. Hinweis: Ein String (.8.) belegt 9 Bytes (einschl. 1 Längenbyte).

c) const MaxKeyLen = 8

2. Aufgabe

Ein Include-File wird durch (*$I Dateiname *) gekennzeichnet.

3. Aufgabe

In der Typdatei müssen der Datensatz (RECORD), MaxDataType und MaxKeyTyp angegeben werden.

4. Aufgabe

```
type str8  = string(. 8.); str15 = string(.15.);
     str17 = string(.17.); str20 = string(.20.);
     AutoSatz = record
                  Satzstatus       : longint; (* 4 *)
                                   (* integer;  * 3 *)
                  KfzNr            : str8;
                  Fabrikat         : str20;
                  Type             : str15;
                  FahrgestellNr    : str17;
                  Anschaffungsjahr : integer;
                end;

     MaxDataType = AutoSatz;
     MaxKeyType  = str8;
```

5. Aufgabe

In der Satzstatusvariablen wird vermerkt, ob der Satz gültig ist. Weist die Varibale einen Wert ungleich Null auf, so gilt der Satz als gelöscht.

6. Aufgabe
Durch die Angabe **uses TACCESS** wird dem Pascal-System mitgeteilt, daß DATABASE verwendet wird.

6.3 Einrichten von Satz- und Indexdatei

Bevor mit Dateien gearbeitet werden kann, müssen die Dateivariablen vereinbart, die Satz- und die Indexdatei auf dem externen Speicher angelegt und geöffnet worden sein. Die dazu erforderlichen Anweisungen – besser: Systemprozeduren – werden im folgenden behandelt.

6.3.1 Dateivereinbarung

Zur Kennzeichnung von Satz- und Indexdatei sind die Datentypen **Datafile** und **Indexfile** zu verwenden. Wenn wir die Dateivariable für die Lager-Satzdatei mit LagerDat, die Dateivariable für die Lager-Indexdatei mit LagerIdx bezeichnen, so lautet die Vereinbarung der Dateivariablen

```
var LagerDat: Datafile:
    LagerIdx: Indexfile:
```

Der Datentyp Datafile kennzeichnet die Satzdateivariable. Der Datentyp für die Indexdateivariable lautet Indexfile.

6.3.2 Erzeugen von Satz- und Indexdatei

MakeFile
Die Satzdatei wird mit Hilfe der Prozedur **MakeFile** auf dem externen Speicher Diskette oder Magnetplatte angelegt. Als Parameter sind anzugeben:

- Name der Satzdateivariablen (logischer Dateiname)
- Name der physischen Datei (physischer Dateiname)
- Größe des Datensatzes

In allgemeiner Form lautet die Anweisung **MakeFile**:

MakeFile (SatzDateiVariable, phys_Dateiname, DatenSatzgröße);

Bezeichnen wir die physische Lager-Satzdatei mit Lager2.Dat, so lautet in unserem Lagerbeispiel die Anweisung MakeFile wie folgt:

MakeFile (LagerDat, 'Lager2.Dat', SizeOf (LagerRec));

Die Funktion **SizeOf** ermittelt, wieviel Bytes der Datensatz LagerRec im Arbeitsspeicher belegt.

MakeIndex

Die Prozedur zum Erzeugen der Indexdatei lautet **MakeIndex**. Als Parameter sind anzugeben:

- Name der Indexdateivariablen (logischer Dateiname)
- Name der physischen Datei (physischer Dateiname)
- Länge des Schlüsselbegriffs
- Schlüsselkennzeichnung:
  In der Schlüsselkennzeichnung wird angegeben, ob die Schlüssel eindeutig sind oder mit doppelten Schlüsseln gearbeitet werden soll. **Eindeutiger** Schlüssel bedeutet, daß jeder Schlüsselbegriff nur einmal in der Indexdatei vorhanden sein darf. Bei doppelten Schlüsseln kann ein Schlüsselbegriff mehrmals in der Indexdatei eingetragen sein. Man spricht dann auch von **mehrdeutigem** Schlüssel.
  Im Turbo-Access-System zeigt eine 0 an dieser Stelle an, daß nur eindeutige Schlüssel zugelassen sind. Durch eine 1 wird festgelegt, daß Schlüssel doppelt vorkommen können.
  In DATABASE wird durch die vordefinierten Konstanten

  NoDuplicates (interner Wert = 0)

  bzw.

  Duplicates (interner Wert = 1)

angezeigt, ob nur eindeutige oder auch doppelte Schlüssel verwendet werden dürfen.
Allgemein lautet die Anweisung zum Erzeugen der Indexdatei:

MakeIndex (IndexDateiVariable, phys_Dateiname, Schlüssellänge, Schlüsselkennzeichnung);

Benennen wir in unserem Lagerbeispiel die physische Indexdatei mit Lager.idx und lassen nur eindeutige Schlüssel zu, so lautet die Anweisung zum Erzeugen der Indexdatei:

MakeIndex (Lageridx, 'Lager.idx', MaxKeyLen,0); (*3*)

bzw.

MakeIndex (Lageridx, 'Lager.idx', MaxKeyLen, NoDuplicates); (*4*)

In den Parametern wird die Systemkonstante MaxKeyLen (maximale Schlüssellänge) angeführt. Der Wert hierfür wurde bereits bei den Angaben zur Strukturierung der Indexdatei festgelegt.

6.3.3 Öffnen von Satz- und Indexdatei

Bevor mit eingerichteten Dateien gearbeitet werden kann, müssen die Dateien geöffnet werden. Dabei werden die für die Verwaltung des Dateisystems erforderlichen Informationen vom externen Speicher in den Arbeitsspeicher geladen.
Das Öffnen der Dateien geschieht mit den Anweisungen **OpenFile** und **OpenIndex**. Die anzugebenden Parameter entsprechen denen, die bei den Anweisungen MakeFile bzw. MakeIndex anzugeben waren. Allgemein lauten die Anweisungen zum Öffnen der Satz- und Indexdatei:

OpenFile (SatzDateiVariable, phys_Dateiname, Datensatzgröße);

Open Index (IndexDateiVariable, phys_Dateiname, Schlüssellänge, Schlüsselkennzeichnung);

Bezogen auf das Lagerbeispiel lauten die Anweisungen zum Öffnen der Satz- und der Indexdatei:

OpenFile (LagerDat, 'Lager2.Dat', SizeOf (LagerRec));
OpenIndex (Lageridx, 'Lager.idx', MaxKeyLen,0); (*3*)

bzw.

OpenIndex (Lageridx, 'Lager.idx', MaxKeyLen, NoDuplicates); (*4*)

6.3.4 Prozeduren zur Anlage und zum Öffnen von Satz- und Indexdatei

Die vorangehenden Ausführungen werden nun auf das Lagerverwaltungsprogramm angewendet.

Hinweis:
Damit Sie die Programmentwicklung am Rechner nachvollziehen können, geben Sie die Variablenvereinbarung, die Prozedur-Auswahl und das Hauptprogramm ein! Diese Programmteile sind ab Seite 211. dargelegt. Entsprechend dem Lernfortschritt fügen Sie die behandelten Prozeduren in das "Rahmenprogramm" ein. Auf diese Weise komplettieren Sie schrittweise das Lagerverwaltungsprogramm.

Das Einrichten und Öffnen der Satzdatei kann man in folgender Prozedur zusammenfassen:

```
procedure SatzDateiOeffnen (var SatzDat : Datafile;
                                PhySDat : str30;
                                Groesse : integer);
begin
  OpenFile (SatzDat, PhySDat, Groesse);
  if not Ok
    then MakeFile (SatzDat,PhySDat,Groesse);
  if not Ok
    then begin
      writeln ('Satzdatei kann nicht geöffnet werden');
      halt;
    end;
end;
```

Die Prozedur wird aufgerufen durch:

SatzDateiOeffnen(Lagerdat,'Lager2.Dat', SizeOf (LagerRec));

Während die **Dateivariable** als **Variablenparameter** an die Prozedur übergeben werden muß, werden die übrigen benötigten Angaben wie Name der physischen Datei und Größe des Datensatzes der Prozedur als Werte zugewiesen.
In der Prozedur SatzDateiOeffnen wird zunächst versucht, die Satzdatei zu öffnen. Wenn diese Operation nicht erfolgreich ausgeführt worden ist, weist die interne Systemvariable Ok den boolschen Wert false auf. Der Grund dafür liegt darin, daß es diese Datei noch nicht gibt. In diesem Fall wird die Datei mittels der MakeFile-Prozedur angelegt. Wenn auch diese Operation nicht ordnungsgemäß ausge-

führt werden konnte, wird der Programmlauf abgebrochen. Der Fehler kann beispielsweise darin liegen, daß auf dem externen Speicher nicht mehr genügend Platz vorhanden ist.

In entsprechender Weise wird die Indexdatei geöffnet bzw. erzeugt.

```
procedure IndexDateiOeffnen (var IndxDat          : Indexfile;
                             PhyIDat          : str30;
                             KeyLaenge, SchlKz : integer);
begin
  OpenIndex (IndxDat, PhyIDat, KeyLaenge, SchlKz);
  if not Ok
    then MakeIndex (IndxDat,PhyIDat,KeyLaenge,SchlKz);
  if not Ok
    then begin
      writeln ('Datei kann nicht geöffnet werden');
      halt;
    end;
end;
```

Diese Form der beiden Prozeduren hat den Vorteil, daß mit jeweils einer Prozedur die Satz- bzw. Indexdatei geöffnet bzw. angelegt werden kann. Ein weiterer Vorteil ist, daß die beiden Prozeduren nahezu allgemeingültig sind. Sie können also auch bei anderen Problemstellungen benutzt werden.

6.3.5 Schließen der Dateien

Bevor ein Programm mit index-sequentiellen Dateien beendet wird, müssen die geöffneten Dateien geschlossen werden. Beim Schließen der Dateien werden die Informationen, die für die Verwaltung des Dateisystem notwendig sind, auf den externen Speicher geschrieben. Unterbleibt das Schließen, besteht die Gefahr, daß das Dateisystem fehlerhaft ist.

Zum Schließen der Dateien werden zwei Anweisungen benötigt. Die Satzdatei wird durch

CloseFile (SatzDateiVariable);

geschlossen. Entsprechend lautet die Anweisung zum Schließen der Indexdatei:

CloseIndex (IndexDateiVariable);

In dem Lagerverwaltungsprogramm lauten somit die Anweisungen zum Schließen der Dateien:

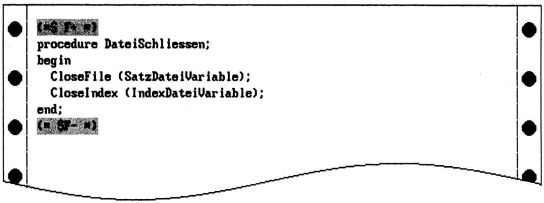
```
CloseFile (Lagerdat);
CloseIndex (Lageridx);
```

Vor Beendigung des Programms müssen geöffnete Dateien unbedingt geschlossen werden, da sie sonst nicht mehr brauchbar sind.

Tritt bei der Entwicklung eines Dateiverwaltungsprogramms ein sogenannter **fataler Fehler** auf, der zum **vorzeitigen Programmabbruch** ohne vorherige Dateischließung geführt hat, so müssen vor dem nächsten Testlauf die angesprochenen Dateien gelöscht werden. Dazu steigen Sie aus dem Turbo-Pascal mittels <Alt> <F> <O> in die Betriebssystemebene um und löschen mittels des Betriebssystemkommandos ERASE die entsprechenden Dateien.
Dieser radikale Weg ist dem Benutzer, der möglicherweise bereits viele Daten eingegeben hat, letztlich nicht zuzumuten. Deshalb sollten die Befehle zum Schließen der Datei in einer Prozedur zusammengefaßt werden, die auch beim Abbruch des Programms noch ausgeführt wird. Dazu müssen wir ein wenig tiefer in das DATABASE-System 4.0 eindringen.
Wenn ein fataler Fehler im Programmlauf aufgetreten ist, kann die Systemroutine TAIOCHECK eine vom Programmierer festgelegte Prozedur aufrufen, bevor sie den Programmlauf abbricht. TAIOCHECK heißt Turbo-Access-Input-Output-Check = Turbo-Access-Eingabe-Ausgabe-Prüfung. Dazu ist erforderlich, daß der Systemvariablen TAErrorProc zu Beginn des Hauptprogramms die interne Speicheradresse der Prozedur mitgeteilt wird, in der die Schließanweisungen enthalten sind. Das geschieht durch die Anweisung

```
begin (*Hauptprogramm *)
TAErrorProc: = @DateiSchliessen;
```

Dem Prozedurnamen wird also das Zeichen @ vorangestellt.
Die Prozedur **DateiSchliessen** ist wie folgt zu gestalten:



```
{* $F+ *}
procedure DateiSchliessen;
begin
  CloseFile (SatzDateiVariable);
  CloseIndex (IndexDateiVariable);
end;
{* $F- *}
```

Vor der Prozedur DateiSchliessen muß die Compiler-Anweisung **(*$F+ *)** gegeben werden, weil bei diesem Prozeduraufruf die aktuelle Unitgrenze übersprungen werden muß. Im Anschluß an die Prozedur muß mit **(* $F− *)** die Standardeinstellung wieder hergestellt werden.

6.3.6 Zusammenfassung

- Der Datentyp der Dateivariablen der Satzdatei lautet **Datafile**. Die Dateivariable der Indexdatei ist vom Typ **Indexfile**.
- Anweisung zum Einrichten einer Satzdatei auf dem externen Speicher:

  MakeFile (SatzDateiVariable, phys_Dateiname, DatenSatzgröße)

- Anweisung zum Öffnen einer Satzdatei:

  OpenFile (SatzDateiVariable, phys_Dateiname, DatenSatzgröße)

- Datentypen der Parameter bei MakeFile und OpenFile:

  SatzDateiVariable: **Datafile**
  phys_Dateiname: **string**
  DatenSatzgröße: **integer**

- Anweisung zum Einrichten einer Indexdatei auf dem externen Speicher:

MakeIndex (SatzDateiVariable, phys_Dateiname, Schlüssellänge, Schlüsselkennzeichnung)

- Anweisung zum Öffnen einer Indexdatei:

  OpenIndex (SatzDateiVariable, phys_Dateiname, Schlüssellänge, Schlüsselkennzeichnung)

- Datentypen der Parameter bei MakeIndex und OpenIndex:

  IndexDateiVariable: **Indexfile**
  phys_Dateiname: **string**
  Schlüssellänge: **integer**
  Schlüsselkennzeichnung: **integer**

- Bei **eindeutigen Schlüsseln** hat die Schlüsselkennzeichnung den Wert **0** bzw. **NoDuplicates**. Bei **mehrdeutigen Schlüsseln** ist der Wert **1** oder **Duplicates**
- Die Dateivariablen für die Satz- und Indexdatei sind bei Prozeduraufrufen als **Variablenparameter** zu übergeben.
- Geöffnete Dateien müssen **unbedingt** wieder geschlossen werden!

  Schließen der Satzdatei: **CloseFile** (SatzDateiVariable)
  Schließen der Indexdatei: **CloseIndex** (IndexDateiVariable)

6.3.7 Übungen

1. Aufgabe
Wie lauten die Typangaben für die Dateivariablen der Satz- und Indexdatei?

2. Aufgabe
Bei den Anweisungen MakeFile und OpenFile ist als ein Parameter die Datensatzgröße anzugeben. Erklären Sie am Beispiel des Kfz-Datensatzes (siehe Aufgabe 1 Kapitel 6.2.4), wie die Datensatzgröße am einfachsten angegeben werden kann!

3. Aufgabe
In den Anweisungen MakeIndex und OpenIndex ist die Länge des Schlüssels anzugeben. Welche Systemkonstante kann hier angegeben werden?

4. Aufgabe
In welcher Anweisung ist anzugeben, daß eine Indexdatei nur eindeutige Schlüssel beinhalten soll?

5. Aufgabe
In welcher Systemvariablen wird vermerkt, ob eine Dateioperation erfolgreich durchgeführt worden ist?

6. Aufgabe
Warum müssen geöffnete Dateien unbedingt wieder geschlossen werden?

6.3.8 Lösungen

1. Aufgabe
SatzDateiVariable: Datafile
IndexDateiVariable: Indexfile

2. Aufgabe
Am einfachsten kann die Datensatzgröße durch SizeOf (Autosatz) angegeben werden.

3. Aufgabe
Die Systemkonstante MaxKeyLen beinhaltet die Schlüssellänge.

4. Aufgabe
Die Schlüsselkennzeichnung auf Eindeutigkeit oder Mehrdeutigkeit erfolgt in den Anweisungen MakeIndex und OpenIndex.

5. Aufgabe
In der Systemvariablen Ok wird vermerkt, ob eine Dateioperation korrekt ausgeführt wurde. Bei korrekter Ausführung weist Ok den Wert true auf. Ok wird auf false gesetzt, wenn bei der Dateioperation Fehler aufgetreten sind.

6. Aufgabe
Beim Schließen werden wichtige Systeminformationen auf den externen Speicher zurückgeschrieben. Wird das Programm ohne vorheri-

ges Schließen der Daten beendet, so gehen die Informationen verloren. Das aber führt zu Fehlern im Dateisystem.

6.4 Eingabe und Ausgabe von Daten bei index-sequentiellen Dateien

6.4.1 Arten von Dateiverarbeitungsoperationen

Zur Verarbeitung von index-sequentiellen Dateien werden folgende Operationen benötigt:

Eintrag eines neuen Datensatzes
Ein neuer Datensatz ist in die Satzdatei zu schreiben. Dessen Schlüssel und Satzadresse ist in die Indexdatei einzutragen. Diese Operationen werden bewirkt durch:

AddRec = Hinzufügen eines Datensatzes
AddKey = Hinzufügen eines Schlüssels

Verändern eines Datensatzes
Über einen Schlüssel soll ein bestimmter Datensatz in den Arbeitsspeicher gelesen werden. Nach der Bearbeitung wird der Datensatz wieder zurückgeschrieben (Update). Dazu werden die Prozeduren

FindKey = Ermitteln der Satzadresse zu einem bestimmten Schlüssel
GetRec = Einlesen eines Datensatzes
PutRec = Zurückschreiben eines Datensatzes

benötigt.

Löschen eines Datensatzes
Datensätze, die nicht mehr gültig sind, sind aus der Datei zu entfernen. Dazu muß der Datensatz wie auch sein Schlüssel gelöscht werden.
Das Löschen eines Schlüsseleintrages und eines Datensatzes wird durch

DeleteKey = Löschen eines Schlüssels
DeleteRec = Löschen eines Datensatzes

bewirkt.

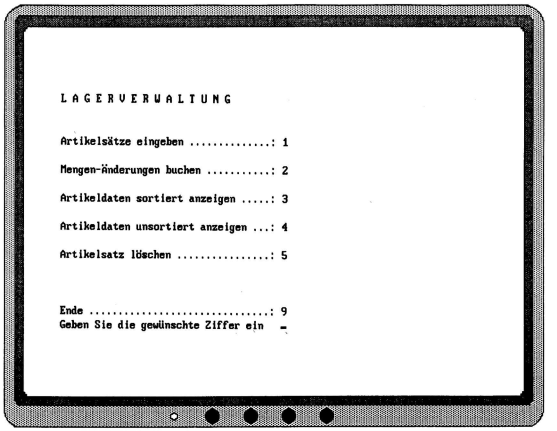
Ermittlung von Satzadressen
Neben dem bereits oben erwähnten FindKey gibt es noch weitere Möglichkeiten, Satzadressen zu ermitteln:

NextKey = Ermitteln der Satzadresse, die zum nächsten Schlüssel gehört. Mittels NextKey wird die Indexdatei sequentiell gelesen.
PrevKey = Ermitteln der Satzadresse, die zum vorhergehenden Schlüssel gehört. Mittels PrevKey wird die Datei rückwärts sequentiell gelesen.
SearchKey = Ermitteln der Satzadresse, deren Schlüssel größer oder gleich einem gegebenen Suchschlüssel ist.
ClearKey = Indexdatei-Zeiger in Grundstellung bringen.

Diese Dateioperationen werden am Beispiel des Lagerverwaltungsprogramms an Hand geeigneter Prozeduren erläutert. Aus diesen Prozeduren wird später das Lagerverwaltungsprogramm zusammengesetzt.
Das Lagerverwaltungsprogramm soll folgenden Leistungsumfang haben:

- Artikeldaten erfassen,
- Mengenveränderungen buchen,
- die Lagerdatei auflisten,
- Artikelsätze löschen.

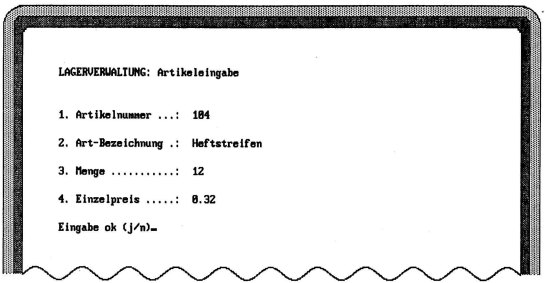
(siehe Abbildung Seite 185)



Zuerst wird die Eingabe von Daten und ihr Eintrag in die Dateien behandelt.

6.4.2 Eingabe und Speicherung von neuen Datensätzen

Problemstellung
Die Prozedur zur Eingabe der Artikeldaten wird so aufgebaut, daß nach der Eingabe der Artikelnummer geprüft wird, ob es diese Artikelnummer bereits im Dateisystem gibt. Ist das der Fall, erfolgt am Bildschirm die Meldung: "Artikelnummer bereits vorhanden". Falls die eingegebene Artikelnummer noch nicht vorhanden ist, werden die weiteren Artikeldaten eingegeben.
Nach Überprüfung der Eingaben auf Richtigkeit wird der Artikelsatz in die Satzdatei geschrieben, und die Artikelnummer wird als Schlüsselbegriff zusammen mit der Satzadresse in die Indexdatei eingetragen.



Wird die anschließende Abfrage auf "weiter?" mit 'j' oder 'J' bestätigt, so können weitere Artikelsätze eingegeben werden. Andernfalls wird die Eingabeprozedur verlassen.

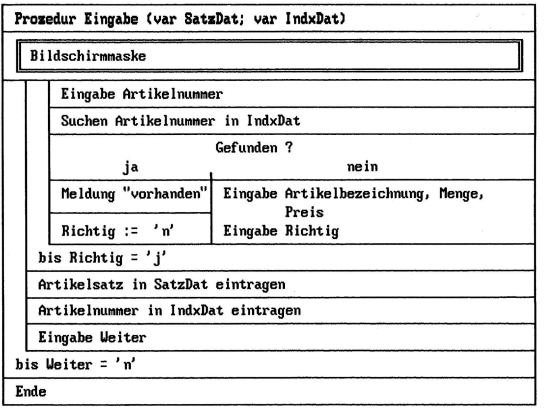
Datensatzaufbau

Der Aufbau des Datensatzes, der bereits bei der Ermittlung der Steuerungsparameter verwendet wurde, sei hier noch einmal aufgezeigt.

```
LagerRec = RECORD
             Recstatus     : Longint; (* 4 *)
                             (* integer; * 3 *)
             Nr            : str5;
             Bezeichnung   : str30;
             Menge         : integer;
             Preis         : real;
           end;
```

Algorithmus

Entsprechend der obigen Anforderung ist der Algorithmus für die Eingabe und Speicherung der Artikeldaten wie folgt zu gestalten:



Programm

```
 1  procedure Eingabe (var SatzDat : Datafile;
                       var IndxDat : Indexfile);
 2  var Weiter, Richtig : char;
 3      SuchSchluessel  : str5;
 4      Satznummer      : longint; (* 4 *)
                         (* integer); * 3 *)
 5      Artikel         : LagerRec;

      procedure Bildschirm;
      begin
        ClrScr;
        Gotoxy (5,3); writeln ('LAGERVERWALTUNG: Artikeleingabe ');
        Gotoxy (5,6); writeln ('1. Artikelnummer ...:');
        Gotoxy (5,8); writeln ('2. Art-Bezeichnung .:');
        Gotoxy (5,10);writeln ('3. Menge ..........:');
        Gotoxy (5,12);writeln ('4. Einzelpreis .....:');
      end;

 6  begin
 7    Bildschirm;
 8    repeat
 9      repeat
10        Gotoxy (27,6); ClrEol; readln (Artikel.Nr);
```

```
11        SuchSchluessel := Artikel.Nr;
12        FindKey (IndxDat, Satznummer,SuchSchluessel);
13        if Ok
14          then begin
15            Gotoxy (5,20);
16            write ('Artikelnummer existiert schon! <RETURN>');
17            Weiter := ReadKey; (*4*) (*3: read (kbd, Weiter); *)
18            Weiter := 'n';
19            Gotoxy (5,20); ClrEol;
20          end
21          else begin
22            Gotoxy (27,8); ClrEol; readln (Artikel.Bezeichnung);
23            Gotoxy (27,10); ClrEol; readln (Artikel.Menge);
24            Gotoxy (27,12); ClrEol; readln (Artikel.Preis);
25            Gotoxy (5,14); ClrEol; write ('Eingabe Ok (j/n)');
26            Richtig := ReadKey; (* 4 *)
27          (* read (kbd, Richtig); * 3 *)
28            Gotoxy (5,14); ClrEol;
29          end;
30      until Richtig in ['j','J'.];
31      Artikel.RecStatus := 0;
32      AddRec (SatzDat, Satznummer, Artikel);
33      AddKey (IndxDat, Satznummer, Artikel.Nr);
34      Gotoxy (5,14); ClrEol; write ('Weiter (j/n)');
35      Weiter := ReadKey; (* 4 *)
    (* read (kbd, Weiter); * 3 *)
36      DelLine;
37    until Weiter in ['n','N'.];
38  end;
```

Die Prozedur wird aufgerufen durch

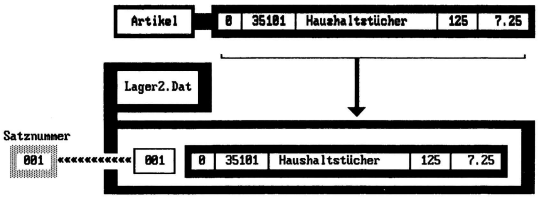
Eingabe (Lagerdat, Lageridx);

Erläuterungen

Eintragen von Datensätzen: Die Anweisung, mit der der Artikelsatz in die Satzdatei eingetragen wird, lautet in Zeile 32

32 AddRec (SatzDat, Satznummer, Artikel);

Dadurch wird der Inhalt von Artikel in die Satzdatei geschrieben, die durch SatzDat adressiert ist. Der Eintrag erfolgt hinter dem bislang letzten Datensatz. Dabei merkt sich das System in der Variablen Satznummer, unter welcher Satzadresse der Satz in die Datei gespeichert wurde. Beim ersten Datensatz ist es die Satzadresse 1, beim zweiten die Satzadresse 2 usw.



Die vor der Artikelnummer stehende Null gibt den Satzstatus an. Er wird mit der Anweisung in Zeile 31

31 Artikel.RecStatus: = 0;

auf Null gesetzt. Die Null besagt, daß der Satz gültig ist.
Allgemein lautet die Anweisung zum Eintragen eines Satzes in die Satzdatei hinter dem bislang letzten Datensatz:

AddRec (SatzDateivariable, Satznummer, Datensatz);

Die Variable Satznummer muß in DATABASE als longint-Variable, im Turbo-Access-System als Integer-Variable vereinbart werden.
Eintragen von Schlüsseln: Mit der Anweisung

33 AddKey (IndxDat, Satznummer, Artikel.Nr)

in Zeile 33 werden die beim AddRec festgelegte Satzadresse sowie der Inhalt von Artikel.Nr als Schlüssel in die Indexdatei eingetragen. Der Eintrag erfolgt aufsteigend sortiert.

Die Anweisung, mit der ein Schlüssel in eine Indexdatei eingetragen wird, lautet allgemein:

AddKey (IndexDateivariable, Satznummer, Schlüssel)

Suchen von Schlüsseln: In der Anweisung

```
12  FindKey (IndxDat, Satznummer, SuchSchluessel);
```

wird die Indexdatei nach dem Schlüssel durchsucht, der in der Variablen SuchSchluessel enthalten ist. Ist der Schlüssel **vorhanden**, wird die Satzadresse in die Variable Satznummer geschrieben und die Systemvariable **Ok** auf **true** gesetzt. Wurde der Schlüssel in der Indexdatei **nicht gefunden**, so erhält die Systemvariable **Ok** den Wert **false**.
In allgemeiner Form lautet die Anweisung zum Suchen eines Schlüssels in der Indexdatei:

FindKey (IndexDateivariable, Satznummer, Suchschlüssel);

Mit der Anweisung AddRec werden die Daten jeweils am Ende der Datei angefügt. Deshalb kann die obige Prozedur Eingabe auch dazu benutzt werden, später weitere Datensätze in die Datei zu schreiben. Durch die vorgeschaltete FindKey-Anweisung wird sichergestellt, daß nur eindeutige Schlüssel angenommen werden.

6.4.3 Ausgabe einer index-sequentiellen Datei

Die in der Lagerdatei gespeicherten Artikelsätze sind am Bildschirm aufzulisten. Zwei Formen von Listprogrammen sind möglich. Zum einen kann die Satzdatei sequentiell gelesen und ausgegeben werden. Zum anderen kann die Satzdatei index-sequentiell unter Verwendung der Indexdatei gelesen und angezeigt werden.

Sequentielles Lesen der Satzdatei
Die Prozedur, mit der die Satzdatei sequentiell gelesen wird, lautet:

```
1  procedure DateiUnsortListen (var SatzDat : Datafile;
                                var IndxDat : Indexfile);
2  var Weiter                  : char;
3      LetzterSatz, Satznummer : longint; (* 4 *)
                               (* integer;  * 3 *)
4      Artikel                 : LagerRec;
5  begin
6    writeln;
7    writeln ('ArtNr   Artikelbezeichnung              Menge   Preis');
8    writeln ('-------------------------------------------------');
9    writeln;
10   LetzterSatz := FileLen (SatzDat) - 1;
11   for Satznummer := 1 to LetzterSatz do
12   begin
13     GetRec (SatzDat, Satznummer, Artikel);
14     if Artikel.RecStatus = 0
15       then with Artikel do
16         writeln (Nr, ' ',Bezeichnung,' ':30-length(Bezeichnung),
17                  Menge:8, Preis:8:2);
18   end;
19   writeln; writeln ('Weiter <RETURN>');
20   Weiter := Readkey; (* 4 *)
   (* read (kbd, Weiter); * 3 *)
21 end;
```

Die Prozedur wird aufgerufen durch

DateiUnsortListen (Lagerdat, Lageridx);

Erläuterungen

Mit der Anweisung **GetRec** (= besorge Datensatz)

13 GetRec (SatzDat, Satznummer, Artikel);

wird das System angewiesen, aus der Satzdatei einen bestimmten Datensatz zu lesen und in die Recordvariable Artikel zu schreiben. Die Satzdatei wird über die Dateivariable SatzDat festgelegt. Der Inhalt von Satznummer bestimmt den zu lesenden Datensatz. Weist die Satznummer den Wert 11 auf, so wird der Satz 11 in der Datei gelesen.

Allgemein lautet die Anweisung zum Lesen in der Satzdatei

GetRec (SatzDateiVariable, Satznummer, Datensatz)

Satznummer ist vom Typ longint (*4*) bzw. integer (*3*).

Bei der Verwendung von GetRec ist darauf zu achten, daß die Record-Variable, in die der Datensatz eingelesen werden soll, in derselben Weise aufgebaut ist wie der Datensatz. Das System führt **keine Typenprüfung** durch. Der Programmierer muß selber auf die notwendige Strukturgleichheit achten. Stimmen Datensatzaufbau und Record-Variable nicht genau überein, werden die Datensätze nicht korrekt verarbeitet!

Die GetRec-Anweisung (Zeile 13) benötigt zur Adressierung des Datensatzes in der Satzdatei stets die **Angabe einer Satznummer**. In dieser Prozedur wird die Adresse des Satzes, der mit GetRec gelesen werden soll, durch die Variable Satznummer festgelegt. Satznummer wird als Laufvariable der FOR-Anweisung (Zeile 11) von 1 bis LetzterSatz hochgezählt. Es werden also alle Sätze vom ersten bis zum letzten Satz sequentiell eingelesen. Ausgegeben werden jedoch nur die Sätze, deren Satzstatus den Wert Null aufweist (Zeile 14 bis 17). Ist der Wert in Artikel.RecStatus ungleich Null, so gilt der Satz als gelöscht. Er wird deshalb nicht angezeigt.
Die Adresse des letzten Satzes wird mit Hilfe der Funktion **FileLen** (= Länge der Datei) ermitteln. In der Anweisung

```
10  LetzterSatz: = FileLen (SatzDat) − 1;
```

wird durch FileLen (SatzDat) festgestellt, aus wie vielen Sätzen die Datei besteht. Dabei bezieht FileLen auch den Satz null mit in die Rechnung ein. Der Satz null enthält jedoch keinen vom Benutzer eingegebenen Datensatz. Er wird vom System zur Speicherung von Steuerinformationen benutzt. Deshalb muß der durch FileLen (SatzDat) ermittelte Wert um 1 vermindert werden.
Die Ausgabe der Datensätze erfolgt in der Reihenfolge, wie sie in der Satzdatei gespeichert worden sind, also in einer nicht sortierten Reihenfolge. Sollen die Datensätze in einer nach dem Schlüssel sortierten Reihenfolge angezeigt werden, so ist das index-sequentielle Lesen anzuwenden.

Index-sequentielles Lesen
Um eine Satzdatei index-sequentiell lesen zu können, muß für die GetRec-Anweisung die Satznummer aus dem ersten Eintrag der Indexdatei bereitgestellt werden. Dazu sind zwei Schritte erforderlich. Im ersten Schritt muß der Zeiger, der den Zugriff auf die Indexdatei

steuert, in die Grundstellung gebracht werden. Das geschieht durch die Anweisung

ClearKey (IndexDateiVariable);

In einem zweiten Schritt werden durch den Befehl

NextKey (IndexDateiVariable, Satznummer, Schlüssel)

die Satznummer und der Schlüssel des ersten Eintrages der Indexdatei in den Arbeitsspeicher gelesen. Damit sind die Informationen bereitgestellt, die für die GetRec-Anweisung benötigt werden. Mittels

GetRec (SatzDateiVariable, Satznummer, Datensatz)

können dann die Daten des ersten Satzes aus der Satzdatei geholt werden. Anschließend werden die Daten auf dem Bildschirm angezeigt. Beim nächsten NextKey-Befehl werden die Satznummer und der Schlüssel des folgenden Datensatzes zur Verfügung gestellt, da NextKey die Indexdatei sequentiell liest.
Nach jedem NextKey-Befehl ist zu prüfen, ob die Systemvariable Ok den Wert true aufweist. Solange Ok true ist, konnte NextKey den Indexdatei-Eintrag lesen. Beinhaltet aber die Systemvariable Ok nach einer NextKey-Anweisung den Wert false, so ist das Ende der Indexdatei erreicht. Alle verfügbaren Sätze sind dann ausgegeben worden.
Aus diesen Überlegungen ergibt sich folgender Algorithmus für das index-sequentielle Lesen:

Algorithmus

```
Prozedur DateiSortListen (var SatzDat; var IndxDat)
Ausgabe Überschrift
Zeiger der IndxDat in Grundstellung bringen
Lesen Satznummer des nächsten Schlüssels          (*Erstlesen*)
  Solange Ok true ist
    Lesen Datensatz gemäß Satznummer
    Ausgabe Datensatz
    Lesen Satznummer des nächsten Schlüssels      (*Folgelesen*)
Ende
```

Programm

```
1  procedure DateiSortListen (var SatzDat : Datafile;
                              var IndxDat : Indexfile);
2  var Weiter     : char;
3      Satznummer : longint; (* 4 *)
                 (* integer;  * 3 *)
4      Artikel    : LagerRec;
5      Schluessel : str5;
6  begin
7    writeln;
8    writeln ('ArtNr   Artikelbezeichnung              Menge   Preis');
9    writeln ('-------------------------------------------------');
10   writeln;
11   ClearKey (IndxDat);
12   NextKey (IndxDat, Satznummer,Schluessel);   (* Erstlesen *)
13   while Ok do
14   begin
15     GetRec (SatzDat, Satznummer, Artikel);
16     with Artikel do
17       writeln (Nr, ' ',Bezeichnung,' ':30-length(Bezeichnung),
18                Menge:8, Preis:8:2);
19       NextKey (IndxDat, Satznummer ,Schluessel); (* Folgelesen *)
20   end;
21   writeln; writeln ('Weiter <RETURN>');
22   Weiter := Readkey; (* 4 *)
   (* read (kbd, Weiter); * 3 *)
23 end;
```

Die Prozedur wird aufgerufen durch

DateiSortListen (Lagerdat, Lageridx);

Erläuterungen

Der Befehl NextKey wird in der Prozedur zweimal gegeben. In Zeile 12 steht er unmittelbar vor dem Beginn der WHILE-Anweisung, in der geprüft wird, ob Ok auf true steht. In Zeile 19 befindet sich die NextKey-Anweisung am Ende der WHILE-Schleife. Als nächste Anweisung wird die WHILE-Anweisung durchgeführt. Durch diese Konstruktion des Erstlesens/Folgelesens wird sichergestellt, daß nach jeder NextKey-Operation unmittelbar geprüft wird, ob das Ende der Index-Datei erreicht ist.

Falls beim NextKey festgestellt wurde, daß kein weiterer Eintrag in der Index-Datei vorhanden ist, wird die Systemvariable Ok auf false

gesetzt. In diesem Fall wird die WHILE-Schleife beendet, und das Programm wird mit der Anweisung in Zeile 21 fortgesetzt.
Die Eingabeanweisung in Zeile 22 hat den Sinn, das Programm so lange anzuhalten, bis eine Taste gedrückt worden ist.
Falls Sie die Satzdatei in absteigender Reihenfolge sortiert ausgeben lassen möchten, brauchen Sie in der obigen Prozedur lediglich NextKey durch PrevKey (= Previous Key = vorhergehender Schlüssel) zu ersetzen. Dann wird der letzte Eintrag in der Indexdatei zuerst gelesen. Bei weiteren PrevKey-Anweisungen wird stets auf den vorhergehenden Eintrag in der Indexdatei zugegriffen.
Allgemein lautet die PrevKey-Anweisung

PrevKey (IndexDateiVariable, Satznummer, Schlüssel);

6.4.4 Zusammenfassung

- Zum Beschreiben und Lesen von index-sequentiellen Dateien werden folgende Anweisungen benötigt. Dabei haben die Anweisungsparameter folgende Datentypen:

SatzDateiVariable	:	**Datafile**
IndexDateiVariable	:	**Indexfile**
Satznummer	:	**integer** (*3*) **longint** (*4*)
Schlüssel	:	**string**
Datensatz	:	**record**

- **AddRec:** Schreibt einen neuen Datensatz aus dem Arbeitsspeicher an das Ende der Satzdatei und speichert die Satzadresse in der Variablen Satznummer.
  AddRec (SatzDateiVariable, Satznummer, Datensatz)
- **AddKey:** Trägt den Schlüssel und die Satzadresse sortiert in die Indexdatei ein.
  AddKey (IndexDateiVariable, Satznummer, Schlüssel)
- **GetRec:** Liest den durch Satznummer adressierten Satz aus der Satzdatei in die RECORD-Variable Datensatz.
  GetRec: (SatzDateiVariable, Satznummer, Datensatz)
- **FindKey:** Findet den Schlüssel in der Indexdatei, der mit dem Suchbegriff in der Variablen Schlüssel übereinstimmt, und speichert die Satzadresse in Satznummer.
  FindKey (IndexDateiVariable, Satznummer, Schlüssel)

Wird der Schlüssel gefunden, weist die Systemvariable Ok den Wert true auf.

- **NextKey:** Liest die Indexdatei **sequentiell** und ermittelt den nächsten Schlüssel und die zugehörige Satzadresse.
  NextKey (IndexDateiVariable, Satznummer, Schlüssel)
  Wird beim sequentiellen Lesen das Ende der Indexdatei erreicht, so weist die Systemvariable Ok den Wert false auf.
- **PrevKey:** Ermittelt beim **sequentiellen** Lesen der Indexdatei den vorhergehenden Schlüssel und die zugehörige Satzadresse.
  PrevKey (IndexDateiVariable, Satznummer, Schlüssel)
  Status von Ok wie bei NextKey.
- **ClearKey:** Setzt den Zeiger der Indexdatei in die Grundstellung.
  ClearKey (IndexDateiVariable)
- Die Funktion **FileLen** (SatzDateiVariable) ermittelt die Anzahl der Datensätze der Satzdatei einschließlich dem vom System benutzten Verwaltungssatz.
- Die Funktion **UsedRecs** (SatzDateiVariable) ermittelt die Anzahl der gültigen Datensätze.

6.4.5 Übungen

1. Aufgabe
In welcher Reihenfolge müssen beim Daten- und Schlüsseleintrag die AddKey- und AddRec-Anweisung stehen?

2. Aufgabe
Welchen Wert hat die boolsche Variable Ok, wenn bei einer FindKey-Anweisung der Suchbegriff nicht gefunden wurde?

3. Aufgabe
Warum muß vor dem sequentiellen Lesen einer Indexdatei zunächst die ClearKey-Anweisung gegeben werden?

4. Aufgabe
Was leistet die Anweisung NextKey?

5. Aufgabe

Entwickeln Sie ein Programm, mit dem eine Kundendatei aufgebaut wird!

Aufbau des Kundensatzes:

```
type str5  = string (.5.);
     str30 = string (.30.);
     KundeRec = record
                  RecStatus : longint; (* 4 *)
                            (* string;   * 3 *)
                  Nummer    : str5;
                  Name      : str30 ;
                  Limit     : real;
                  Kredit    : real;
                end;
```

Kredit ist der Betrag, den der Kunde dem Unternehmen schuldet. Unter Limit ist die Obergrenze des Kredits zu verstehen, den das Unternehmen seinem Kunden einräumt.

Die Kundenummer wird als Schlüssel verwendet.

a) Erstellen Sie die Prozeduren zum Anlegen und Öffnen der Satz- und Indexdatei!

b) Formulieren Sie die Prozedur zur Eingabe der Kundendaten! Folgende Verarbeitungsbedingungen sind zu beachten:

- Die Eingabedaten sind der nachstehenden Abbildung des Bildschirms zu entnehmen:

```
LAGERVERWALTUNG: Kunden-Eingabe

1. Kunden-Nummer ...:  60235

2. Kunden-Name .....:  Autohaus Mailering

3. Kredit-Grenze ...:  5000.00

4. Kredit ..........:  1225.00

Eingabe ok (j/n)_
```

- Nach der Eingabe der Kundennummer ist zuerst zu prüfen, ob die Kundennummer bereits in der Indexdatei enthalten ist.
- Die Richtigkeit der Eingaben ist zu bestätigen. Ist die Eingabe korrekt, werden die Daten gespeichert. Ist die Eingabe nicht in Ordnung, ist die gesamte Eingabe zu wiederholen.
- Nach jeder Speicherung kann der Benutzer wählen, ob er die Arbeit fortsetzen möchte. Falls nicht, soll das Auswahlmenü angezeigt werden.

c) Formulieren Sie eine Prozedur, mit der die Kundendaten, sortiert nach Kundennummern, am Bildschirm angezeigt werden!

d) Entwickeln Sie eine Prozedur, mit der die Namen der Kunden angezeigt werden, die den ihnen eingeräumten Kredit überzogen haben (Kredit> Limit). Die Liste darf unsortiert sein.

e) Schreiben Sie eine Prozedur, in der das Programm-Menü angezeigt wird!

f) Schreiben Sie das Hauptprogramm!

6.4.6 Lösungen

1. Aufgabe
Die AddRec-Anweisung muß vor der AddKey-Anweisung stehen, da die Satzadresse, die in die Indexdatei eingetragen wird, bei der Ausführung der AddRec-Anweisung festgestellt wird.

2. Aufgabe
Wurde der Schlüsseleintrag nicht gefunden, weist Ok den Wert false auf.

3. Aufgabe
Vor dem sequentiellen Lesen muß der Satzzeiger der Indexdatei in die Grundstellung gebracht werden. Das bewirkt die ClearKey-Anweisung.

4. Aufgabe
NextKey liest den jeweils nächsten Schlüssel und die zugehörige Satzadresse.

5. Aufgabe
(siehe gegenüberliegende Seite)

```
program Kundendateiverwaltung;
uses CRT, TACCESS;
(*$I Kunde.Typ *)
var KundeDat : Datafile;
    KunNrIdx : Indexfile;
    Wahl     : char;
    Ende     : boolean;


procedure SatzDateiOeffnen (var SatzDat : Datafile;
                                PhySDat : str30;
                                Groesse : integer);
begin
  OpenFile (SatzDat, PhySDat, Groesse);
  if not ok
    then MakeFile (SatzDat, PhySDat, Groesse);
  if not ok
    then begin
      writeln ('Datei kann nicht geöffnet werden');
      halt;
    end;
end;


procedure IndexDateiOeffnen ( var IndxDat          : Indexfile;
                                  PhyIDat          : str30;
                                  KeyLaenge, SchlKz : integer);
begin
  OpenIndex (IndxDat, PhyIDat, KeyLaenge, SchlKz);
  if not ok
    then MakeIndex (IndxDat,PhyIDat,KeyLaenge,SchlKz);
  if not ok
    then begin
      writeln ('Datei kann nicht geöffnet werden');
      halt;
    end;
end;


procedure Eingabe (var SatzDat : Datafile;
                   var IndxDat : Indexfile);
var Weiter, Richtig : char;
    SuchSchluessel  : str5;
    Satznummer      : longint;
    Kunde           : KundeRec;

  procedure Bildschirm;
  begin
    ClrScr;
    Gotoxy (5,3); writeln ('KUNDENVERWALTUNG: Kunden-Eingabe ');
    Gotoxy (5,6); writeln ('1. Kunden-Nummer ...:');
    Gotoxy (5,8); writeln ('2. Kunden-Name .....:');
```

```
    Gotoxy (5,10);writeln ('3. Kredit-Grenze ...:');
    Gotoxy (5,12);writeln ('4. Kredit .........:');
  end;

begin
  Bildschirm;
  repeat
    repeat
      Gotoxy (27,6); ClrEol; readln (Kunde.Nummer);
      SuchSchluessel := Kunde.Nummer;
      FindKey (IndxDat, Satznummer,SuchSchluessel);
      if ok
        then begin
          Gotoxy (5,20);
          write ('Kundennummer existiert schon! <RETURN>');
          Weiter := ReadKey; (* 4 *)
       (* read (kbd, Weiter)  * 3 *)
          Richtig := 'n';
          Gotoxy (5,20); ClrEol;
        end
        else begin
          Gotoxy (27,8); ClrEol; readln (Kunde.Name);
          Gotoxy (27,10); ClrEol; readln (Kunde.Limit);
          Gotoxy (27,12); ClrEol; readln (Kunde.Kredit);
          Gotoxy (5,14); ClrEol; write ('Eingabe ok (j/n)');
          Richtig := ReadKey; (* 4 *)
       (* read (kbd, Richtig); * 3 *)
          Gotoxy (5,14); ClrEol;
        end;
    until Richtig in ('j','J');
    Kunde.RecStatus := 0;
    AddRec (SatzDat, Satznummer, Kunde);
    AddKey (IndxDat, Satznummer,Kunde.Nummer);
    Gotoxy (5,14); ClrEol; write ('Weiter (j/n)');
    Weiter := ReadKey; (* 4 *)
 (* read (kbd, Weiter); * 3 *)
    DelLine;
  until Weiter in ('n','N');
end;

procedure DateiAnzeigen (var SatzDat : Datafile;
                         var IndxDat : Indexfile);
var Weiter                : char;
    SatzAnzahl, Satznummer : longint;
    Kunde                 : KundeRec;
    SuchSchluessel        : str5;
begin
  ClrScr;
  writeln;
  writeln ('Kd-Nr   Kunden-Name                    Limit      Kredit');
  writeln ('-------------------------------------------------------');
  writeln;
  ClearKey (IndxDat);
```

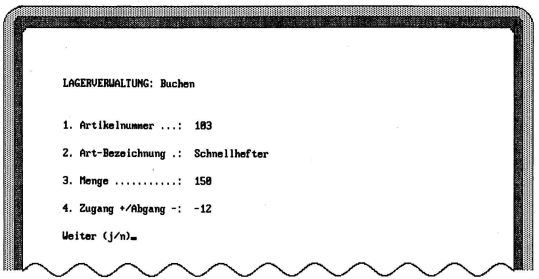
```
    NextKey (IndxDat, Satznummer, SuchSchluessel);
    while ok do
    begin
      GetRec (SatzDat, Satznummer, Kunde);
      with Kunde do
        writeln (Nummer, ' ',Name,' ':30-length(Name),
                 Limit:10:2, Kredit:10:2);
      NextKey (IndxDat, Satznummer, SuchSchluessel);
    end;
    writeln; writeln ('Weiter <RETURN>');
    Weiter := Readkey;  (* 4 *)
  (* read (kbd, Weiter); * 3 *)
  end;


  procedure Ueberziehungsliste (var SatzDat : Datafile);
  var Weiter                : char;
      SatzAnzahl, Satznummer : longint;
      Kunde                 : KundeRec;
  begin
    ClrScr;
    Gotoxy (5,20);
    writeln ('U E B E R Z I E H U N G S L I S T E');
    writeln;
    SatzAnzahl := FileLen (SatzDat);
    for Satznummer := 1 to SatzAnzahl -1 do
    begin
      GetRec (SatzDat, Satznummer, Kunde);
      if (Kunde.RecStatus = 0) and (Kunde.Kredit > Kunde.Limit)
        then writeln (Kunde.Nummer:6,' ',Kunde.Name);
    end;
    writeln; write ('weiter <RETURN>');
    Weiter := ReadKey;  (* 4 *)

    (* read (kbd, Weiter); * 3 *)
    end;


    procedure Auswahl (var Menu : char);
    begin
      ClrScr;
      Gotoxy (5,5); writeln ('K U N D E N V E R W A L T U N G');
      Gotoxy (5,08); writeln ('Kundensätze eingeben ...........: 1');
      Gotoxy (5,10); writeln ('Kundendaten anzeigen ............: 2');
      Gotoxy (5,12); writeln ('Überziehungsliste ..............: 3');
      Gotoxy (5,20); writeln ('Ende ............................: 9');
      Gotoxy (5,21); write ('Geben Sie die gewünschte Ziffer ein ');
      Menu := ReadKey; (* read (kbd, Menu); * 3 *)
    end;
```

```
begin                                        (* HAUPTPROGRAMM *)
  SatzDateiOeffnen (Kundedat, 'Kunde.dat', SizeOf(KundeRec) );
  IndexDateiOeffnen (KunNrIdx, 'KunNr.idx',MaxKeyLen, NoDuplicates);
  Ende := false;
  repeat
    Auswahl (Wahl);
    case Wahl of
      '1' : Eingabe (Kundedat, KunNrIdx);
      '2' : DateiAnzeigen (Kundedat, KunNrIdx);
      '3' : Ueberziehungsliste (Kundedat);
      else  Ende := true;
    end;
  until Ende;
  CloseFile (Kundedat);
  CloseIndex (KunNrIdx);
end.
```

6.5 Verändern und Löschen von Datensätzen

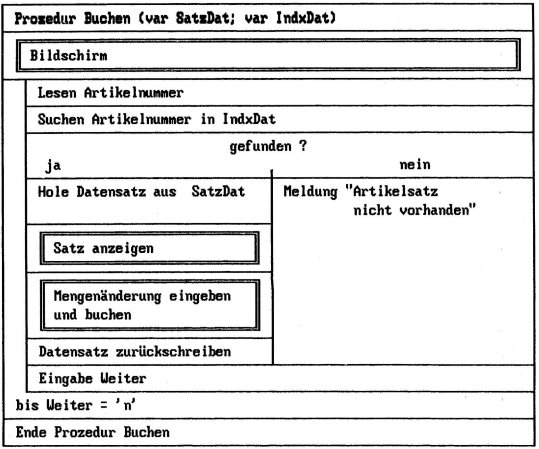
6.5.1 Verändern von Datensätzen (Update)

Problemstellung
Das Verändern von gespeicherten Datensätzen wird am Beispiel der Verbuchung von Mengenänderungen dargestellt.



Die Prozedur Buchen ist in der Weise zu gestalten, daß die Nummer des Artikels eingegeben wird, bei dem sich die Lagermenge verändert hat. An Hand der Artikelnummer ist in der Indexdatei zu prüfen, ob der Artikel existiert. Wenn nicht, erfolgt ein entsprechender Hinweis am Bildschirm.
Ist die Artikelnummer vorhanden, wird der Artikelsatz eingelesen und auf dem Bildschirm ausgegeben. Nachdem die Mengenänderung gebucht worden ist, ist der geänderte Satz wieder in die Datei zurückzuschreiben.

Algorithmus
Bei dieser Vorgehensweise ist der Algorithmus wie folgt zu gestalten:



Programm

```
1  procedure Buchen (var SatzDat : Datafile;
                     var IndxDat : Indexfile);
2  var Weiter         : char;
3      SuchSchluessel : str5;
4      Satznummer     : longint; (* 4 *)
                      (* integer;  * 3 *)
5      Aenderung      : integer;
6      Artikel        : LagerRec;

   procedure Bildschirm;
   begin
     ClrScr;
     Gotoxy (5,3); writeln ('LAGERVERWALTUNG: Buchen');
     Gotoxy (5,6); writeln ('1. Artikelnummer ...:');
     Gotoxy (5,8); writeln ('2. Art-Bezeichnung .:');
     Gotoxy (5,10);writeln ('3. Menge ..........:');
     Gotoxy (5,12);writeln ('4. Zugang +/Abgang -:');
     Gotoxy (5,14);write   ('Weiter (j/n)');
   end;

   procedure SatzAnzeigen;
   begin
```

```
    Gotoxy (27,8); ClrEol; writeln (Artikel.Bezeichnung);
    Gotoxy (27,10); ClrEol; writeln (Artikel.Menge:6);
  end;

  procedure MengenEingabe;
  begin
    Gotoxy (27,12); ClrEol; readln (Aenderung);
    Artikel.Menge := Artikel.Menge + Aenderung;
    Gotoxy (27,10); writeln (Artikel.Menge:6);
  end;

7 begin
8   Bildschirm;
9   repeat
10    Gotoxy (27,6); ClrEol; readln (SuchSchluessel);
11    FindKey (IndxDat, Satznummer, SuchSchluessel);
12    if Ok
13      then begin
14        GetRec (SatzDat, Satznummer, Artikel);
15        SatzAnzeigen;
16        Mengeneingabe;
17        PutRec (SatzDat, Satznummer, Artikel);
18      end
19      else begin
20        Gotoxy (5,20);
21        writeln ('Artikelsatz nicht vorhanden <RETURN>');
22        Weiter := ReadKey; (* 4 *)
        (* read (kbd, Weiter); * 3 *)
23        Gotoxy (5,20); ClrEol;
24      end;
25    Gotoxy (20,14);
26    Weiter := ReadKey; (* 4 *)
    (* read (kbd, Weiter); * 3 *)
26   until Weiter in ['n','N'];
27 end;
```

Die Prozedur wird aufgerufen durch

Buchen (Lagerdat, Lageridx);

Erläuterungen

Mit der uns schon bekannten Anweisung

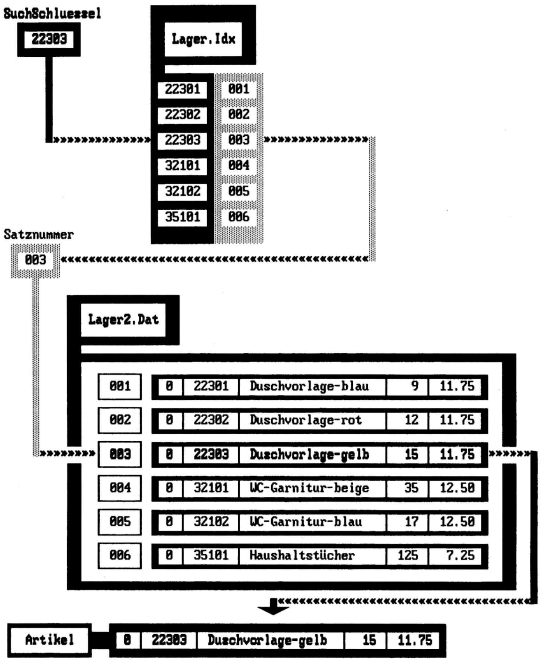
11 FindKey (IndxDat, Satznummer, SuchSchluessel);

wird in Zeile 11 die Satznummer des Datensatzes ermittelt, dessen Artikelnummer als Schlüssel in die Variable SuchSchluessel eingegeben worden war. Falls der Schlüssel nicht gefunden wurde, enthält die Systemvariable Ok den Wert false.

Wurde der Schlüssel gefunden, so wird in der Variablen Satznummer die Satzadresse bereitgestellt.
Durch die Anweisung

```
14   GetRec (SatzDat, Satznummer, Artikel);
```

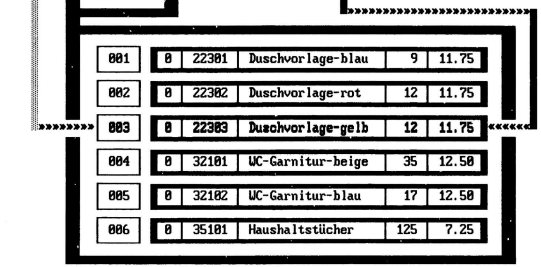
wird der durch Satznummer adressierte Artikelsatz in den Arbeitsspeicher gelesen und steht in der Recordvariablen Artikel zur weiteren Verarbeitung zur Verfügung.



Nachdem der Abgang von 3 gelben Duschvorlagen gebucht worden ist, wird der geänderte Datensatz in die Satzdatei zurückgeschrieben. Das geschieht durch die Anweisung

```
17  PutRec (SatzDat, Satznummer, Artikel);
```

Mit **PutRec** wird das System angewiesen, den Inhalt der Recordvariablen Artikel auf den Satz in der Datei zurückzuschreiben, der durch Satznummer adressiert ist. Wenn in der Variablen Satznummer der Wert 3 enthalten ist, erfolgt die Schreiboperation auf Satz 3 der Satzdatei.
Es ergibt sich dann folgendes Bild:



Während mit GetRec ein bestimmter Datensatz eingelesen werden kann, wird mit PutRec ein Datensatz auf einen bestimmten Platz in der Datei geschrieben.
Allgemein lautet die PutRec-Anweisung:

PutRec (SatzDateivariable, Satznummer, Datensatz);

Datensatz ist eine Record-Variable. Satznummer weist den Datentyp longint (*4*) bzw. integer (*3*) auf.
Das Zurückschreiben des geänderten Satzes muß auf die alte Adresse erfolgen. Sonst würde die Indexdatei nicht mehr zur Satzdatei passen.

Das Dateisystem wäre dann nicht mehr konsistent, in sich stimmig. Deshalb muß beim PutRec die Satznummer denselben Wert aufweisen wie beim GetRec.

6.5.2 Löschen von Datensätzen

Zum Löschen eines Datensatzes sind zwei Anweisungen notwendig. Zunächst muß mit der Anweisung

DeleteKey (IndexDateiVariable, Satznummer, Schlüssel)

der Schlüsseleintrag in der Indexdatei gelöscht werden. Dabei wird in der Variablen Satznummer die Adresse des zu löschenden Satzes festgehalten. Diese Information wird für den zweiten Schritt benötigt. Mittels

DeleteRec (SatzDateiVariable, Satznummer)

wird der Datensatz in der Satzdatei gelöscht. Dabei trägt das System automatisch in das Satzstatusfeld einen Wert ein, der ungleich Null ist. Dadurch wird es dem System ermöglicht, beim Eintrag eines neuen Satzes durch AddRec den durch Löschen frei gewordenen Platz neu zu belegen.
Die einfachste Form des Löschens eines Datensatzes besteht darin, daß die Artikelnummer des zu löschenden Datensatzes eingegeben wird. Anschließend wird mit DeleteKey der Schlüssel und mit DeleteRec der Datensatz gelöscht.

Einfache Löschprozedur

```
procedure Loeschen;
begin
  write ('Eingabe der Artikelnummer des zu löschenden Artikels');
  readln (SuchSchluessel);
  DeleteKey (IndxDat, Satznummer, SuchSchluessel);
  DeleteRec (SatzDat, Satznummer);
end;
```

Diese Löschprozedur ist jedoch gefährlich. Es besteht die Gefahr, daß aus Versehen ein falscher Datensatz gelöscht wird. Deshalb ist es sinnvoll, sich den zu löschenden Datensatz anzeigen zu lassen, bevor die Löschoperation in der Datei durchgeführt wird.

Abgesicherte Löschprozedur

Die mit "Sicherung" versehene Prozedur zum Löschen eines Datensatzes lautet:

```
1  procedure DatensatzLoeschen (var SatzDat : Datafile;
                                var IndxDat : Indexfile);
2  var Satznummer      : longint; (* 4 *)
                       (* integer;  * 3 *)
3      SuchSchluessel  : str5;
4      Antwort, Weiter : char;
5      Artikel         : LagerRec;
6  begin
7    ClrScr;
8    Gotoxy (5,6); writeln ('LAGERVERWALTUNG: Datensatz löschen');
9    Gotoxy (5,12); writeln ('Artikelnummer eingeben ..:');
10   Gotoxy (5,14); writeln ('Artikelbezeichnung .....:');
11   repeat
12     gotoxy (27,12); ClrEol; readln (SuchSchluessel);
13     FindKey (IndxDat, Satznummer, SuchSchluessel);
14     if Ok
15       then begin
16         GetRec (SatzDat, Satznummer, Artikel);
17         Gotoxy (27,14); ClrEol; writeln (Artikel.Bezeichnung);
18         Gotoxy (5,16); write ('Löschen? (j/n)');
19         Antwort := ReadKey; (* 4 *)
20       (* read (kbd, Antwort); * 3 *)
21         Gotoxy (5,16); ClrEol;
22         if Antwort in ['j','J']
23           then begin
24             DeleteKey (IndxDat, Satznummer, SuchSchluessel);
25             DeleteRec (SatzDat, Satznummer);
26           end;
27       end
28       else begin
29         Gotoxy (5,16);
30         write ('Satznummer nicht vorhanden <RETURN>');
31         Antwort := ReadKey; (* 4 *)
32       (* read (kbd, Antwort); * 3 *)
33       end;
34     Gotoxy (5,16); write ('weiter? (j/n)');
35     Weiter := ReadKey;  (* 4 *)
36   (* read (kbd, Weiter); * 3 *)
     DelLine;
   until Weiter in ['n','N'];
end;
```

Die Prozedur wird aufgerufen durch

```
DatensatzLoeschen (Lagerdat, Lageridx);
```

Erläuterungen
In dieser Prozedur wird in den Zeilen 7 bis 10 zunächst eine kleine Bildschirmmaske aufgebaut. In der REPEAT-Schleife, die von Zeile 11 bis 35 reicht, können beliebig viele Artikelsätze gelöscht werden, bis in der Eingabeanweisung (Zeile 33) die Taste <N> betätigt wird.
In Zeile 12 wird die Artikelnummer des zu löschenden Artikelsatzes in die Variable SuchSchluessel eingegeben. Mit Hilfe der FindKey-Anweisung in Zeile 13 wird geprüft, ob die eingegebene Artikelnummer in der Indexdatei existiert. Wenn sie vorhanden ist, die Systemvariable Ok also den Wert true aufweist, wird der Artikelsatz gelesen und die Artikelbezeichnung auf dem Bildschirm angezeigt (Zeilen 16 und 17).
Nun kann der Benutzer entscheiden, ob der Satz gelöscht werden soll. Gibt er ein 'j' oder 'J' in die Variable Antwort ein (Zeile 19), so wird der Eintrag in der Indexdatei und Satzdatei gelöscht (Zeilen 23 und 24). Tippt er jedoch einen anderen Buchstaben ein, erfolgt keine Löschung.
Falls beim FindKey (Zeile 13) die Artikelnummer nicht gefunden wurde, wird am Bildschirm ein entsprechender Hinweis ausgegeben (Zeilen 27 bis 31).

6.5.3 Zusammenstellung der Prozeduren zum Lagerverwaltungsprogramm

Die für die Verwaltung der Lagerdatei erforderlichen Prozeduren sind damit geschaffen. Das Lagerverwaltungsprogramm wird nachfolgend in einer verkürzten Form dargestellt:

```
program Lagerverwaltung2;
uses CRT, TACCESS;  (* 4 *)
(*$I Lager.typ *)   (* 4 *)
var LagerDat : Datafile;
    Lageridx : Indexfile;
    Wahl     : char;
    Ende     : boolean;


procedure SatzDateiOeffnen  (var SatzDat : Datafile;
                                 PhySDat : str30;
                                 Groesse : integer);
  ...
procedure IndexDateiOeffnen (var IndxDat           : Indexfile;
                                 PhyIDat           : str30;
                                 KeyLaenge, SchlKz : integer);
  ...
procedure Eingabe           (var SatzDat : Datafile;
                                 var IndxDat : Indexfile);
  ...
procedure Buchen            (var SatzDat : Datafile;
                                 var IndxDat : Indexfile);
  ...
procedure DateiSortListen   (var SatzDat : Datafile;
                                 var IndxDat : Indexfile);
  ...
procedure DateiUnsortListen (var SatzDat : Datafile;
                                 var IndxDat : Indexfile);
  ...
procedure DatensatzLoeschen (var SatzDat : Datafile;
                                 var IndxDat : Indexfile);
  ...


procedure Auswahl (var Menu : char);
begin
  ClrScr;
  Gotoxy (5, 5); writeln ('L A G E R V E R W A L T U N G');
  Gotoxy (5,08); writeln ('Artikelsätze eingeben ..............: 1');
  Gotoxy (5,10); writeln ('Mengen-Änderungen buchen ...........: 2');
  Gotoxy (5,12); writeln ('Artikeldaten sortiert anzeigen .....: 3');
  Gotoxy (5,14); writeln ('Artikeldaten unsortiert anzeigen ...: 4');
  Gotoxy (5,16); writeln ('Artikelsatz löschen ................: 5');
  Gotoxy (5,20); writeln ('Ende ...............................: 9');
  Gotoxy (5,21); write ('Geben Sie die gewünschte Ziffer ein');
  Menu := ReadKey;  (* 4 *)
(* read (kbd, Menu); * 3 *)
end;
```

```
begin                                        (* HAUPTPROGRAMM *)
  InitIndex;          (* 3 *)
  SatzDateiOeffnen(Lagerdat, 'Lager.dat', SizeOf(LagerRec) );
  IndexDateiOeffnen(Lageridx, 'Lager.idx', MaxKeyLen);
  Ende := false;
  repeat
    Auswahl (Wahl);
    case Wahl of
      '1' : Eingabe (Lagerdat, Lageridx);
      '2' : Buchen (Lagerdat, Lageridx);
      '3' : DateiSortListen (Lagerdat, Lageridx);
      '4' : DateiUnsortListen (Lagerdat, Lageridx);
      '5' : DatensatzLoeschen (Lagerdat, Lageridx);
      else  Ende := true;
    end;  (* of case *)
  until Ende;
  CloseFile (Lagerdat);
  CloseIndex (Lageridx);
end.
```

Bei Verwendung der DATABASE zur Version 3.x sind die zweite und dritte Zeile des obigen Programms durch folgende Vereinbarungen zu ersetzen:

```
const MaxDataRecSize = 47;
      MaxKeyLen      =  5;
      PageSize       = 24;
      Order          = 12;
      PageStackSize  =  8;
      MaxHeight      =  5;
(*$I ACCESS3.BOX *)          (* Kein Semikolon! *)
(*$I GETKEY.BOX *)
(*$I ADDKEY.BOX *)
(*$I DELKEY.BOX *)
type str30    = string(.30.);
     str5     = string (.5.);
     LagerRec = RECORD
                  Satzstatus    : integer
                  Nr            : str5;
                  Bezeichnung   : str30;
                  Menge         : integer;
                  Preis         : real;
                end;
```

6.5.4 Zusammenfassung

Folgende Anweisungen werden zum Verändern und Löschen von Datensätzen benötigt:

- **PutRec:** Schreibt einen Datensatz zurück in die Satzdatei.
  PutRec (SatzDateiVariable, Satznummer, Datensatz)
- **DeleteKey:** Löscht einen durch den Schlüssel bestimmten Eintrag in der Indexdatei und speichert die zugehörige Satzadresse in der Variablen Satznummer.
  DeleteKey (IndexDateiVariable, Satznummer, Schlüssel)
- **DeleteRec:** Löscht den durch Satznummer adressierten Datensatz und verändert automatisch den Satzstatus.
  DeleteRec (SatzDateiVariable, Satznummer)
- Zu den Datentypen der Parameter in den obigen Anweisungen siehe Kapitel 6.4.4.
- Im Turbo-Access-System ist durch **InitIndex** das Indexsystem zu initialisieren.
- Weitere Anweisungen in DATABASE
  Um häufige Disketten- bzw. Plattenzugriffe zu vermeiden, die die Verarbeitungsgeschwindigkeit verringern, werden Veränderungen in Satz- und Indexdatei nicht unmittelbar auf den externen Datenträger geschrieben, sondern im Arbeitsspeicher gehalten. Bei einem Systemzusammenbruch sind deshalb die letzten Änderungen in der Regel nicht auf den externen Speicher geschrieben worden. Das unverzügliche Speichern auf dem externen Speicher kann durch

  FlushFile (Satzdateivariable)
  FlushIndex (IndexDateiVariable)

  vom Anwenderprogramm erzwungen werden.
- Mit den Anweisungen

  EraseFile (SatzDateiVariable)
  EraseIndex (IndexDateiVariable)

  werden Satz- und Indexdatei gelöscht.

Bevor die Dateilöschanweisungen durchgeführt werden, sollte das Programm um eine Bestätigung bitten, damit nicht Dateien versehentlich gelöscht werden.

6.5.5 Übungen

1. Aufgabe
Gegeben ist folgender Programmausschnitt, mit dem ein Datensatz gelesen, verändert und zurückgeschrieben werden soll. Nehmen Sie die erforderlichen Ergänzungen vor!

```
  ...
readln (SuchSchluessel);
FindKey (InxDat, SatzNr, SuchSchluessel);
_______
  then  begin
          ____________________________
          Eingabe_Aenderung;
          ____________________________
          end
  else writeln ('Schluessel nicht gefunden');
  ...
```

2. Aufgabe
Ein Datensatz ist zu löschen. Dazu wird folgender Programmausschnitt vorgegeben. Nehmen Sie die erforderlichen Ergänzungen vor!

```
  ...
readln (SuchSchluessel)
_____________________________
if Ok then
  begin
  ____________________________
  ____________________________
  end
  else writeln ('Schluessel nicht vorhanden')
  ...
```

3. Aufgabe
Das obige Kundenverwaltungsprogramm (siehe Seite 197) ist um folgende Funktionen zu ergänzen:

a) Zahlungen von Kunden vermindern den gewährten Kredit. Verkäufe gegen Rechnung erhöhen ihn. Entwickeln Sie eine Prozedur, mit der der beanspruchte Kredit fortgeschrieben wird!

b) Kundendaten sind zu löschen. Formulieren Sie eine "abgesicherte" Löschprozedur!
c) Nehmen Sie die erforderlichen Ergänzungen in der Prozedur Programmauswahl und im Hauptprogramm vor!

6.5.6. Lösungen

1. Aufgabe

```
...

readln (SuchSchluessel)
FindKey (IndxDat, SatzNr, SuchSchluessel);
if Ok
  then begin
    GetRec (SatzDat, SatzNr, Datsatz);
    Eingabe_Aenderung;
    PutRec (SatzDat, SatzNr, Datsatz);
  end
  else writeln ('Schluessel nicht gefunden');

...
```

2. Aufgabe

```
...

readln (SuchSchluessel)
FindKey (IndxDat, SatzNr, SuchSchluessel);
if Ok
  then begin
    DeleteKey (IndxDat, SatzNr, SuchSchluessel);
    DeleteRec (SatzDat, SatzNr);
  end
  else writeln ('Schlüssel nicht vorhanden');

...
```

3. Aufgabe

```
procedure Buchen (var SatzDat : Datafile;
                  var IndxDat : Indexfile);
var Weiter         : char;
    SuchSchluessel : str5;
    Satznummer     : longint;
    Aenderung      : real;
    Kunde          : KundeRec;

  procedure Bild1;
  begin
    ClrScr;
    Gotoxy (13,3); writeln ('KUNDENVERWALTUNG: Buchen ');
    Gotoxy (5,6); writeln ('1. Kunden-Nummer .....:');
    Gotoxy (5,8); writeln ('2. Kunden-Name .....:');
    Gotoxy (5,10);writeln ('3. Kredit .....:');
    Gotoxy (5,12);writeln ('4. Rechnung + / Zahlung - .:');
    Gotoxy (5,14); write ('Weiter (j/n)');
  end;

  procedure SatzAnzeigen;
  begin
    Gotoxy (33,8); ClrEol; writeln (Kunde.Name);
    Gotoxy (33,10); ClrEol; writeln (Kunde.Kredit:10:2);
  end;

  procedure MengenEingabe;
  begin
    Gotoxy (33,12); ClrEol; readln (Aenderung);
    Kunde.Kredit := Kunde.Kredit + Aenderung;
    Gotoxy (33,10); writeln (Kunde.Kredit:10:2);
  end;

begin
  Bild1;
  repeat
    Gotoxy (33,6); ClrEol; readln (SuchSchluessel);
    FindKey (IndxDat, Satznummer, SuchSchluessel);
    if ok
      then begin
        GetRec (SatzDat, Satznummer, Kunde);
        SatzAnzeigen;
        Mengeneingabe;
        PutRec (SatzDat, Satznummer, Kunde);
      end
      else begin
        Gotoxy (5,20);
         write ('Kundensatz nicht vorhanden <RETURN>');
        Weiter := ReadKey; (* read (kbd, Weiter); * 3 *)
        DelLine;
      end;
```

```
      Gotoxy (28,14); Weiter := ReadKey; (* read (kbd, Weiter); * 3 *)
  until Weiter in ['n','N'];
end;


procedure DatensatzLoeschen (var SatzDat : Datafile;
                             var IndxDat : Indexfile);
var Satznummer      : longint;
    SuchSchluessel  : str5;
    Antwort, Weiter : char;
    Kunde           : KundeRec;

begin
  ClrScr;
  Gotoxy (5,6); writeln ('KUNDENVERWALTUNG: Kundensatz löschen');
  Gotoxy (5,12); writeln ('Kunden-Nummer eingeben ..:');
  Gotoxy (5,14); writeln ('Kunden-Name ............:');
  repeat
    gotoxy (33,12); ClrEol; readln (SuchSchluessel);
    FindKey (IndxDat, Satznummer, SuchSchluessel);
    if ok
     then begin
       GetRec (SatzDat, Satznummer, Kunde);
       Gotoxy (33,14); ClrEol; writeln (Kunde.Name);
       Gotoxy (5,16); write ('Löschen? (j/n)');
       Antwort := ReadKey; (* read (kbd, Antwort); * 3 *)
       Gotoxy (5,16); ClrEol;
       if Antwort in ['j','J']
         then begin
           DeleteKey (IndxDat, Satznummer, SuchSchluessel);
           DeleteRec (SatzDat, Satznummer);
         end;
     end
     else begin
       Gotoxy (5,16);
       write ('Satznummer nicht vorhanden <RETURN>');
       Antwort := ReadKey; (* read (kbd, Antwort); * 3 *)
     end;
    Gotoxy (5,16); write ('weiter? (j/n)');
    Weiter := ReadKey; (* read (kbd, Weiter); * 3 *)
    DelLine;
  until Weiter in ['n','N'];
end;


procedure Auswahl (var Menu: char);
begin

  ...

  Gotoxy (5,14); writeln ('Rechnungen / Zahlungen buchen ..: 4');
  Gotoxy (5,16); writeln ('Kundensatz löschen .............: 5');
  Gotoxy (5,28); writeln ('Ende ...........................: 9');
```

```
  Gotoxy (5,21); write ('Geben Sie die gewünschte Ziffer ein ');
  Menu := ReadKey; (* read (kbd, Menu); * 3 *)
end;

begin                                    (* HAUPTPROGRAMM *)

  ...

    '4' : Buchen (Kundedat, KunNrIdx);
    '5' : DatensatzLoeschen (Kundedat, KunNrIdx);
    else  Ende := true;
  end;

  ...
```

6.6 Behandlung von Dateien mit nicht eindeutigem Schlüssel

Einführungsbeispiel: Die Mitarbeiter in der Lagerverwaltung möchten, daß das Programm Lagerverwaltung2 noch um eine Suchfunktion erweitert wird. Ein Mitarbeiter formuliert die Anforderung an das Programm wie folgt: "Ein Kunde ruft an und fragt 'Was habt ihr an Duschvorlagen da, und wie teuer sind diese?' Dann möchte ich nur **dusch** am Rechner eingeben, und der Computer soll mir alle Artikel anzeigen, die mit **dusch** beginnen."
Zur Lösung dieses Problems muß eine zweite Indexdatei geschaffen werden. Da in dem angeforderten Programmteil die Satzdatei auf bestimmte Artikelbezeichnungen abgefragt werden soll, ist der Schlüssel der zweiten Indexdatei aus der Artikelbezeichnung zu bilden. Es soll hier als ausreichend angesehen werden, wenn der Schlüssel aus den ersten 5 Zeichen der Artikelbezeichnung gebildet wird.

6.6.1 Anlegen und öffnen der Indexdatei

Da es mehrere Artikel geben kann, deren Bezeichnung in den ersten 5 Zeichen übereinstimmt, werden die Schlüssel der zweiten Indexdatei **nicht eindeutig** sein. Das muß beim Anlegen und Öffnen dieser Indexdatei bei der Schlüsselkennzeichnung berücksichtigt werden.
Die Dateivariable für die zweite Indexdatei wird mit **Bezeichidx** festgelegt. Die physische Indexdatei wird mit **LagBez.idx** bezeichnet.

Wenn wir die obige Prozedur **IndexDateiOeffnen** (siehe Kapitel 6.3.4) verwenden, so sind der Prozedur folgende aktuellen Parameter zu übergeben:

	formaler Parameter	aktueller Parameter
Name der Indexdateivariablen:	IndxDat	Bezeichidx
Name der physischen Datei :	PhyIDat	LagBez.idx
Schlüssellänge :	KeyLaenge	MaxKeyLen
Schlüsselkennzeichnung :	SchlKz	Duplicates (*4*)
		1 (*3*)

Die Prozedur **IndexDateiOeffnen** wird aus dem Hauptprogramm aufgerufen durch:

(*4*): IndexDateiOeffnen (Bezeichidx, 'LagBez.idx', MaxKeyLen, **Duplicates**);
(*3*): IndexDateiOeffnen (Bezeichidx, 'LagBez.idx', MaxKeyLen, **1**);

Da Bezeichidx als globale Dateivariable benötigt wird, muß sie im Vereinbarungsteil zum Hauptprogramm vermerkt werden:

```
var LagerDat: Datafile;
    Lageridx, Bezeichidx: Indexfile;
```

Durch die zweite Indexdatei ergeben sich keine Änderungen in der Behandlung der Satzdatei.

6.6.2 Schlüsseleintrag in die Indexdatei

Um den 5stelligen Artikelbezeichnungs-Schlüssel in die Indexdatei LagBez.idx einzutragen, wird die Prozedur **Eingabe** (siehe Seite 187/188) um einige Anweisungen erweitert.
Der 5stellige Schlüssel wird aus der Artikelbezeichnung abgeleitet, indem die ersten 5 Zeichen der Artikelbezeichnung in die Variable SchlName kopiert werden:

```
SchlName := copy (Artikel.Bezeichnung,1,5)
```

Damit dem Benutzer des Programms später keine Probleme durch die der Groß- oder Kleinschreibung des Suchbegriffes entstehen, wird

der Inhalt von SchlName grundsätzlich in Großbuchstaben dargestellt. Dazu wird die Funktion **Upcase** verwendet, die einen Kleinbuchstaben (außer Umlaute: ä, ü, ö) in einen Großbuchstaben umwandelt. Die Umwandlung in Großbuchstaben geschieht durch:

```
for I := 1 to 5 do SchlName (.I.) := upcase (SchlName (.I.));
```

Der Eintrag des so gebildeten Schlüssels in die Indexdatei, die in der Prozedur Eingabe durch LogIBez addressiert wird, geschieht durch die Anweisung

```
AddKey (LogIBez, Satznummer, SchlName);
```

Nun sind noch die neuen Variablen im Vereinbarungsteil der Prozedur zu deklarieren und der Prozedurkopf um die weitere Indexdatei zu ergänzen.
Der Prozedurkopf lautet nun:

```
procedure Eingabe (var SatzDat: Datafile; var IndxDat: Indexfile;
var IndBezDat: Indexfile);
```

In der Variablenvereinbarung sind folgende Ergänzungen vorzunehmen:

```
var Weiter, Richtig: char;
    SchlName, ZwiNr: str5;
    Satznummer: longint; (*4*) (*3: integer; *)
    Artikel: LagerRec;
    I : integer;
```

Damit bei Artikelbezeichnungen, die weniger als 5 Zeichen umfassen, später keine Probleme beim Bilden und Finden von Schlüsseln auftreten, wird vor die Eingabeanweisung

```
23  readln (Artikel.Bezeichnung);
```

eine Anweisung geschaltet, mit der die Variable in ihrer ganzen Länge mit Leerzeichen belegt wird. Diese Anweisung lautet:

```
FillChar (Artikel.Bezeichnung, SizeOf (Artikel.Bezeichnung, ' ');
```

Die Anweisungen zur Schlüsselbildung und zum Schlüsseleintrag in die Indexdatei IndBezDat werden hinter die AddKey-Anweisung der Prozedur Eingabe eingefügt:

```
32  AddRec (SatzDat, Satznummer, Artikel);
33  AddKey (IndxDat, Satznummer, Artikel.Nr);
    SchlName := copy (Artikel.Bezeichnung,1,5);
    for I := 1 to 5 do SchlName(.I.) := upcase (SchlName(.I.));
    AddKey (IndBezDat, Satznummer, SchlName);
```

Schließlich ist der Aufruf der Prozedur Eingabe im Hauptprogramm (siehe Seite 212) um die Angabe der zweiten Indexdatei zu ergänzen:

'1': Eingabe (LagerDat, Lageridx, **Bezeichidx**);

Mit diesen Änderungen ist das Programm nun in der Lage, zur Lagerdatei eine zweite Indexdatei zu führen, in der ein nicht eindeutiger Schlüssel, gebildet aus den ersten 5 Zeichen der Artikelbezeichnung, eingetragen wird.

6.6.3 Suchen in einer Indexdatei mit nicht eindeutigem Schlüssel

Der Rechner soll die Artikelsätze anzeigen, deren Schlüssel mit einem eingegebenen Suchbegriff übereinstimmt. Wurde beispielsweise **dusch** oder **Dusch** eingegeben, so sollen alle Artikel angezeigt werden, deren Artikelbezeichnung mit **Dusch** beginnt.
Für unsere weiteren Überlegungen wird angenommen, daß die Indexdatei, die über IndBezDat adressiert wird, folgenden Inhalt hat:

Schlüssel	Satznummer
DUSCH	001
DUSCH	004
DUSCH	005
DURCH	002
SEIFE	003
SEIFE	006
EOF	

Die Lösung zu diesem Problem wollen wir im Wege der schrittweisen Verfeinerung ableiten. In einem ersten Ansatz kann man den Algorithmus wie folgt skizzieren:

```
Beginn
Eingabe SuchBegriff
Suche in der Indexdatei nach dem ersten Schlüssel, der größer oder
gleich SuchBegriff ist
while OK
    Lesen Datensatz
    Ausgabe Datensatz
    Lesen nächster Schluessel
Ende
```

Gibt man in der Anweisung Eingabe (Schluessel) **DUSCH** als Suchbegriff ein, so wird mit der Suchanweisung der erste Schlüssel, der auf DUSCH lautet, in der oben dargestellten Indexdatei gefunden und die Satznummer 001 ermittelt. In der WHILE-Schleife wird der Artikelsatz gelesen und ausgegeben. Mit der folgenden Schlüssellesen-Anweisung wird der nächste Schlüssel, der ebenfalls auf DUSCH lautet, gelesen und die Satznummer 004 ermittelt. Der entsprechende Artikelsatz wird gelesen und ausgegeben.
Die drei Artikelsätze mit dem Schlüssel DUSCH werden in richtiger Weise gelesen und angezeigt. Dann jedoch gibt das System alle weiteren Artikelsätze aus, weil das Ende der Datei noch nicht erreicht ist. Die WHILE-Schleife wird erst dann verlassen, wenn der Wert in Ok wegen Datei-Ende auf false steht. Damit die WHILE-Schleife schon vorher beendet wird, muß die Eintrittsbedingung erweitert werden.
Wie aus der Darstellung der Indexdatei zu erkennen ist, sind die Schlüsseleinträge sortiert angeordnet. Deshalb kann es beim sequentiellen Durchsuchen der Indexdatei keinen weiteren Eintrag DUSCH geben, wenn bei einer NextKey-Anweisung ein Schlüssel eingelesen wurde, der ungleich DUSCH ist. Somit ist die WHILE-Schleife auch dann zu verlassen, wenn der zuletzt gelesene Schlüssel nicht mehr mit dem eingegebenen Suchbegriff übereinstimmt.
Die Eintrittsbedingung der WHILE-Anweisung muß also lauten:

```
while Ok und SuchSchluessel = SuchBegriff
```

Damit diese Vergleichsoperation durchgeführt werden kann, muß der Inhalt von SuchBegriff in eine zweite Variable, hier SuchSchluessel genannt, kopiert werden. Die Variable SuchSchluessel wird dann in

den Anweisungen SearchKey und NextKey als Schlüsselvariable verwendet.
Schließlich muß in dem obigen Grob-Algorithmus noch berücksichtigt werden, daß die Schlüsseleinträge stets groß geschrieben worden sind. Nach der Eingabe des Suchbegriffes müssen seine Buchstaben in Großbuchstaben umgesetzt werden.
In Form des Struktogramms lautet der verfeinerte Algorithmus:

Prozedur ArtikelSuchen (var SatzDat; var IndBezDat)
Eingabe SuchBegriff
SuchBegriff in Großbuchstaben umwandeln
SuchSchuessel := SuchBegriff
Suche ersten Schlüssel in IndBezDat, der größer oder gleich SuchSchluessel ist
Solange nicht IndBezDat-Ende und SuchSchluessel = SuchBegriff
Lesen Artikelsatz
Ausgabe Artikelsatz
Lesen nächster Schlüssel
Ende

Programm

```
procedure ArtikelSuchen (var SatzDat   : Datafile;
                         var IndBezDat : Indexfile);
var Weiter                 : char;
    Satznummer             : longint; (* 4 *)
                           (* integer;  * 3 *)
    SuchBegriff, SuchSchluessel : str5;
    I                      : integer;
    Artikel                : LagerRec;
begin
  ClrScr;
  Gotoxy (5,20);
  write ('Gesuchter Artikel (5 Anfangsbuchstaben): ');
  FillChar (Suchbegriff, SizeOf (Suchbegriff),' ');
  readln (SuchBegriff);
  for I := 1 to 5 do SuchBegriff (.I.) := upcase (SuchBegriff(.I.));
  SuchSchluessel := SuchBegriff;
  SearchKey (IndBezDat, Satznummer, SuchSchluessel);
  while Ok and (SuchBegriff = SuchSchluessel) do
  begin
    GetRec (SatzDat, Satznummer, Artikel);
    with Artikel do
      writeln (Nr, ' ',Bezeichnung,' ':30-length(Bezeichnung),
               Menge:8, Preis:8:2);
    NextKey (IndBezDat, Satznummer, SuchSchluessel);
  end;
  writeln; writeln ('Weiter <RETURN>'); Weiter := Readkey;
(* read (kbd, Weiter); * 3 *)
end;
```

Die Prozedur ist aus dem Hauptprogramm durch

ArtikelSuchen (Lagerdat, Bezeichidx);

aufzurufen.
Um die Prozedur in das Programm Lagerverwaltung2 einzubinden, müssen in der Prozedur Auswahl und im Hauptprogramm (siehe Seite 211/212) Ergänzungen vorgenommen werden.
In der Prozedur Auswahl ist einzufügen:

Gotoxy (5,18); writeln, ('Artikel suchen / anzeigen...:6');

Die CASE-Struktur des Hauptprogramms ist zu ergänzen um

'6': ArtikelSuchen (LagerDat, Bezeichidx);

Schließlich ist am Ende des Programms die Anweisung zum Schließen der zweiten Indexdatei einzufügen:

```
...
CloseIndex (Lageridx);
CloseIndex (Bezeichidx);
end.
```

Mit dieser Programmerweiterung wurde ein Lagerverwaltungsprogramm geschaffen, das eine Lager-Satzdatei und zwei Indexdateien verwaltet.

6.6.4 Zusammenfassung

- Eine Indexdatei mit mehrdeutigem Schlüssel hat die Schlüsselkennzeichnung **Duplicates** oder **1**.
- **SearchKey**
  Sucht den ersten Schlüssel in der Indexdatei, der gleich oder größer als der Suchbegriff in der Variablen Schlüssel ist. Die zugehörige Satzadresse wird in Satznummer gespeichert.
  SearchKey (IndexDateiVariable, Satznummer, Schlüssel)
- Besteht ein Schlüssel aus Buchstaben, so sollte der Schlüssel stets in Großbuchstaben in die Indexdatei eingetragen werden.
- Vor eine Eingabe von Suchbegriffen sollte die Eingabevariable mittels
  FillChar (SchlVar, SizeOf (SchlVar), ' ');
  mit Leerzeichen beschrieben werden.

6.6.5 Übungen

1. Aufgabe
In einer Indexdatei mit mehrdeutigen Schlüsseln sind folgende Angaben enthalten:

Schluessel	Satzadresse	
MUELL	008	Welche Satzadresse ermittelt die
SCHNE	005	Anweisung SearchKey, wenn als
SCHNE	017	Suchbegriff "SCHNE" vorgegeben
SCHNE	006	ist?
SCHUS	009	

2. Aufgabe
Die Löschprozedur in dem obigen Lagerverwaltungsprogramm muß erweitert werden, da nun auch der Schlüssel des zu löschenden Artikels in der Indexdatei Bezeich.idx ausgetragen werden muß.
Gestalten Sie diese Erweiterung als Prozedur, an die die Satz- und Indexdatei als Variablenparameter, der Suchbegriff (die ersten 5 Zeichen in Großbuchstaben) und die Satznummer aus der eindeutigen Schlüsseldatei als Werteparameter übergeben werden!
Hinweis: Bei Indexdateien mit mehrdeutigen Schlüsseln muß vor dem DeleteKey ein GetRec auf die Satzdatei durchgeführt worden sein.

6.6.6 Lösungen

1. Aufgabe
SearchKey sucht den ersten Schlüssel, der gleich oder größer als der Suchbegriff ist. In diesem Fall wird die Satzadresse 005 eingelesen. Hätten sie SAMSO als Suchbegriff vorgegeben, so hätte SearchKey ebenfalls die Satzadresse 005 geliefert, weil "SCHNE" der erste Schlüssel ist, bei dem gilt: "SCHNE" > = "SAMSO".

2. Aufgabe

```
procedure DatensatzLoeschen (var SatzDat   : Datafile;
                             var IndxDat   : Indexfile;
                             var IndBezDat : Indexfile);
var Satznummer             : longint;
    SuchSchluessel, SuSchl : str5;
    Antwort, Weiter        : char;
    Artikel                : LagerRec;
    I                      : integer;

  procedure LoeschDops (var SatDat  : Datafile;
                        var DopIndx : Indexfile;
                            Satnr   : longint;
                            SuchKey : str5);
  var SatzAdr    : longint;
      SuchSchl   : str5;
      ZwiArtikel : LagerRec;
  begin
    SuchSchl := SuchKey;
    SearchKey (DopIndx, SatzAdr, SuchSchl);
    while Ok and (SuchSchl = SuchKey) do
    begin
      if SatNr = SatzAdr
        then begin
          GetRec (SatDat, SatzAdr, ZwiArtikel);
          DeleteKey (DopIndx, SatzAdr, SuchSchl);
        end;
      NextKey (DopIndx, SatzAdr, SuchSchl);
    end;
  end;

begin
  ClrScr;
  Gotoxy (5,6); writeln ('LAGERVERWALTUNG: Datensatz löschen');

  ...

      Gotoxy (5,16); ClrEol;
      if Antwort in ('j','J')
        then begin
          SuSchl := copy (Artikel.Bezeichnung,1,5);
          for I := 1 to 5 do
            SuSchl(.I.) := upcase(SuSchl (.I.));
          LoeschDops (SatzDat,IndBezDat, Satznummer, SuSchl);
          DeleteKey (IndxDat, Satznummer, SuchSchluessel);

  ...

end;
```

7 DATENSTRUKTUR TEXTDATEI

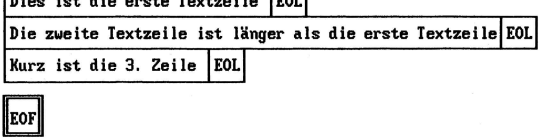
Einführungsbeispiel: Die Datenerfassung am Bildschirm verlangt gute Bildschirmmasken. Nach unserem jetzigen Kenntnisstand müssen wir sie noch Schritt für Schritt programmieren. Das bedeutet bei umfangreichen Masken, wie beispielsweise der nachstehenden Erfassungsmaske, einen großen Programmieraufwand.

```
                    Hotel Vredener Hof

                  Zimmer-Nummer: <  >

1. Hausname: <                                          >
2. Vorname : <                                          >

3. Strape  : <                                          >
4. Wohnort : <                                          >

5. Übernachtung von          6. Übernachtungen bis
   Tag  Monat  Jahr             Tag  Monat  Jahr
   <  ><  >  <  >               <  ><  >  <  >

          Weiter (j/n):
```

Die Mühe, solche Bildschirmmasken zu programmieren, bleibt uns jedoch erspart, weil wir sie in relativ einfacher Weise über eine Textdatei in ein Programm einbringen können.

7.1 Charakteristik der Datenstruktur Textdatei

Eine Textdatei enthält beliebig viele Textzeilen, die unterschiedlich lang sein dürfen.



Eine Textzeile kann aus beliebigen Zeichen bestehen. Das Ende einer Zeile wird durch die **EOL**-Marke angezeigt (EOL = End Of Line = Ende der Zeile). Das Ende der Datei wird durch die EOF-Marke gekennzeichnet (EOF = End of File).

7.2 Erstellen einer Textdatei

Um die obige Bildschirmmaske als Textdatei zu erstellen, brauchen Sie kein eigenes Programm zu entwickeln. Verwenden Sie dazu den Editor des Turbo-Pascal-Systems! Schließen Sie jede Zeile mit 'RETURN' ab! Dadurch wird die EOL-Marke an die gerade geschriebene Zeile angehängt. Die Datei-Endemarke wird automatisch gesetzt.
Beim Speichern des eingegebenen Textes sollten Sie jedoch eine eigene Dateikennung angeben, da das Pascal-System sonst automatisch ein .PAS an den Dateinamen anhängt. Um deutlich zu machen, daß es sich um eine Bildschirmmaske handelt, könnte der Dateiname Bild 01.msk lauten.
Sie können die Textdatei aber auch mit Hilfe eines selbsterstellten Programmes aufbauen:

```
1  program TextErfassung;
2  var BildZeile : string(.255.);
3      TextDatei : text;
4  begin
5    assign(TextDatei, 'Bild01.msk');
6    rewrite (TextDatei);
7    repeat
8      readln (BildZeile);
9      writeln (TextDatei, BildZeile);
10   until eof;              (* Abfrage auf Eingabe <CTRL> <Z> *)
11   close (TextDatei);
12 end.
```

Erläuterungen

In Turbo-Pascal wird eine Datei vom Typ TEXT mit Hilfe des Schlüsselwortes **Text** vereinbart (siehe Zeile 3). Allgemein lautet die Vereinbarung:

var Dateivariable: text;

Die Zuordnung zwischen der Dateivariablen und der physischen Datei auf dem externen Datenspeicher geschieht, wie aus Zeile 5 zu ersehen ist, durch die ASSIGN-Anweisung. In allgemeiner Formulierung:

assign (Dateivariable, Name_physische_Datei);

Die Regeln für den Aufbau der ASSIGN-Anweisung sind uns schon aus dem Kapitel 5.2.2 bekannt.
Eine Datei vom Typ Text wird mit

rewrite (Dateivariable)

zum Schreiben geöffnet. In diese Datei kann, anders als bei den satzorientierten Dateien, **nur geschrieben werden**.
Die Datei wird mittels der CLOSE-Anweisung geschlossen:

close (Dateivariable);

Die Eingabe des Textes und das Speichern erfolgt in der REPEAT-Anweisung:

```
 7   repeat
 8     readln (BildZeile);
 9     writeln (TextDatei, BildZeile);
10   until eof;
```

In dieser REPEAT-Schleife können nun beliebig viele Zeilen eingegeben werden. Jede Zeile wird durch <RETURN> abgeschlossen. Anschließend speichert die WRITELN-Anweisung den in die Variable Zeile eingetippten Text in die Datei. Die Schleife wird verlassen, wenn **<CTRL> <Z>** <RETURN> eingegeben wurde. Diese Tastenkombination erzeugt den Code EOF.
Allgemein lautet die Anweisung zum Speichern eines Textes:

writeln (Dateivariable, StringVariable);

Geben Sie das Programm ein und testen Sie es! Dabei werden Sie sicher auch feststellen, daß es einfacher ist, die Textdatei mit Hilfe des Editors von Turbo-Pascal zu erzeugen. Vor allem ist bei der Erstellung von Bildschirmmasken der Vorteil nicht zu unterschätzen, daß man zum einen den Entwurf der Maske vor dem Einsatz im Programm auf Übersichtlichkeit überprüfen kann. Zum anderen kann man aus der Darstellung am Bildschirm die Spalten- und Zeilenangaben für die Eingabefelder ablesen. Schließlich muß man bei Fehlern oder Änderungen nicht die gesamte Eingabe wiederholen.

7.3 Lesen einer Textdatei

Bevor Sie sich mit dem nächsten Programm auseinandersetzen, sollten Sie mit Hilfe des Editors die obige Bildschirmmaske erstellen und unter dem Namen Bild01.msk speichern. Sie können dann diese Textdatei durch das nachstehende Programm vom externen Speicher in den Arbeitsspeicher einlesen und am Bildschirm anzeigen lassen.
Ein Programm zum Lesen einer Textdatei und zur Ausgabe am Bildschirm ist im Prinzip wie folgt aufgebaut:

```
1  program Bildschirmmaske_Lesen;
2  uses CRT;        (* 4 *)
3  var BsMaske: text;
4        BildZeile: string (.255.);

5  begin
6    assign (BsMaske, 'Bild01.msk');
7    reset (BsMaske);
8    ClrScr;
9    while not eof (BsMaske) do
10   begin
11     readln (BsMaske, BildZeile);
12     writeln (BildZeile);
13   end;
14   close (BsMaske)
15 end.
```

Erläuterungen

Die Vereinbarung der Dateivariablen (hier: BsMaske) und die Zuordnung zur physischen Datei erfolgen nach der uns schon bekannten Regel:

assign (Dateivariable, Name_physische_Datei);

Eine Textdatei wird zum Lesen durch die RESET-Anweisung geöffnet:

reset (Dateivariable);

Im Gegensatz zu den satzorientierten Dateien kann eine durch RESET geöffnete Textdatei **nur gelesen werden**.
Eine Textdatei wird geschlossen durch

close (Dateivariable);

Eine Textdatei wird zeilenweise gelesen. Die Anweisung dazu lautet allgemein:

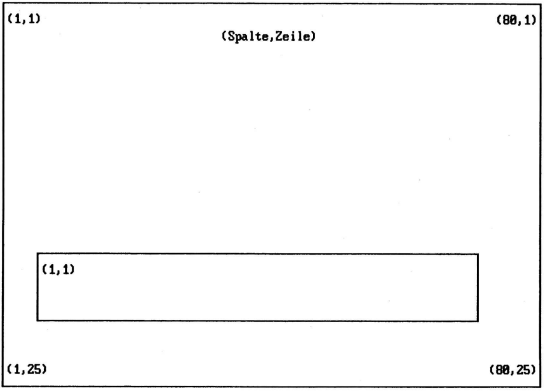
readln (Dateivariable, StringVariable),

Für das zeilenweise Lesen der Textdatei und die Anzeige der Textzeile auf dem Bildschirm ist eine WHILE-Schleife erforderlich (Zeile 9 bis 13). Diese beiden Anweisungen werden ausgeführt, solange das EOF-Merkmal in der Datei nicht erkannt wurde.

7.4 Einbeziehung von Fehlertexten – Fenstertechnik

Zu einem gut durchdachten Eingabeprogramm gehören neben einer übersichtlichen Bildschirmmaske auch Fehlertexte, die den Benutzer auf Eingabefehler aufmerksam machen. Die Fehlertexte können in einem **Fenster (window)** dargestellt werden.

Charakteristik eines Fensters
Unter einem Fenster versteht man einen Bildschirmausschnitt.



Dieser Bildschirmausschnitt verhält sich wie ein eigenständiger Bildschirm. Wurde ein Fenster eröffnet und werden aus einer Textdatei Texte zeilenweise eingelesen, so erfolgt die Ausgabe nur in diesem Bildschirmausschnitt. Umfaßt der Text mehr Zeilen als das Fenster, so werden die ersten Zeilen aus dem Fenster "herausgeschoben". Da der übrige Teil des Bildschirms nicht ansprechbar ist, verschwinden sie über dem oberen Fensterrand. Der Text rollt gewissermaßen durch das Fenster.
Die Eigenständigkeit des Bildschirmfensters läßt sich auch daran er-

kennen, daß sich der Befehl ClrScr, mit dem sonst der gesamte Bildschirm gelöscht wird, nur auf den Bildschirmausschnitt bezieht. Der übrige Teil des Bildschirms außerhalb des Fensters bleibt von dem Befehl ClrScr unberührt. Auch die Cursor-Steuerung bezieht sich bei geöffnetem Fenster nur auf das Fenster. Ein Gotoxy (1,1), der sonst den Cursor in die linke obere Bildschirmecke stellt, positioniert bei eingerichtetem Fenster den Cursor in die linke obere Fensterecke.

Einrichten eines Fensters

Ein Fenster wird durch die Standardprozedur **window** eingerichtet. Wollen Sie beispielsweise ein Fenster in den Zeilen 18 bis 21 und den Spalten 5 und 70 anlegen, so müssen Sie folgende Anweisung geben:

```
window (5,18,70,21).
```

Im WINDOW-Befehl werden die Zeilen- und Spaltenangaben in folgender Reihenfolge vermerkt:

window (von_Spalte,von_Zeile,bis_Spalte,bis_Zeile)

Die Zeilen- und Spaltenangaben müssen vom Typ INTEGER sein.
In ein Fenster können weitere Fenster eingeschachtelt werden.
Um ein Fenster wieder aufzuheben, wird der WINDOW-Befehl erneut angewendet, nun aber mit den Koordinaten des gesamten Bildschirms:

```
window (1,1,80,25)
```

Durch diese Anweisung wird der gesamte Bildschirm wieder als Fenster aktiviert.
Fenstertechnik und Textdateien werden nun in einem Programm, in dem es um die Behandlung von Eingabefehlern geht, angewendet.
Sie haben sicher beim Testen der bisherigen Programme die Erfahrung gemacht, daß Programme "abgestürzt" sind, weil Sie in eine INTEGER-Variable einen REAL-Wert oder einen Buchstaben eingegeben haben. Deshalb wird nun ein Programm entwickelt, das zum einen den sogenannten Formatfehler abfängt. Zum anderen soll es nur ganzzahlige Werte zwischen 1 und 12 zulassen. Ein Formatfehler liegt vor, wenn beispielsweise in eine INTEGER-Variable ein REAL-Wert eingegeben wurde.

Bei einer fehlerhaften Eingabe soll das Programm einen der beiden nachstehenden Fehlertexte anzeigen. Geben Sie die beiden Fehlertexte mit Hilfe des Editors ein, und speichern Sie diese jeweils als eigene Dateien unter den angegebenen Dateinamen!
Die Fehlertexte sollen lauten:

Fehler 01.txt

```
Fehler in der Eingabe: Es sind nur ganze Zahlen zugelassen.
Wiederholen Sie die Eingabe! <RETURN>
```

Fehler 02.txt

```
Fehler in der Eingabe:
Es sind nur Zahlen zwischen 1 und 12zugelassen!  <RETURN>
```

Um überprüfen zu können, ob ein eingegebener Wert das richtige Format hat, müssen wir den Compilerbefehl (*$I-*)verwenden. Mit diesem Compilerbefehl (*$I-*) wird der automatische Programmabbruch bei einer fehlerhaften Eingabe-/Ausgabeoperation ausgeschaltet. Ob bei der Eingabe ein Formatfehler aufgetreten ist, wird über die Funktion IOResult festgestellt. Hier soll der von IOResult gelieferte Wert in einer Variablen Fehler gespeichert werden. Weist Fehler einen Wert ungleich Null auf, so stimmt der Datentyp des Eingabewertes nicht mit dem Datentyp der Eingabevariablen überein. In diesem Fall ist in einem Fenster die Fehlermeldung auszugeben, die in der Textdatei Fehler01.txt gespeichert ist.

Der Algorithmus in Form des Pseudocodes:

```
Wiederhole
  E/A-Prüfung abschalten
  Eingabe (Zahl)
  Fehler := IOResult
  Wenn Fehler <> 0
        Dann Fenster einrichten
             Datei Fehler01.txt lesen
             Text am Bildschirm anzeigen
             Fenster löschen
bis Fehler = 0
```

Ferner ist laut Aufgabenstellung zu prüfen, ob die eingegebene Zahl innerhalb des zulässigen Bereiches liegt. Dies wird mit Hilfe einer Auswahl-Anweisung erreicht.

```
Wenn nicht (Zahl in (.1..12.))
      Dann Fehler := 1
           Fenster einrichten
           Datei Fehler02.txt lesen
           Text am Bildschirm anzeigen
           Fenster löschen
```

Die Anweisungen auf den beiden Dann-Wegen sind nahezu identisch. Sie unterscheiden sich nur darin, daß unterschiedliche Dateien einzulesen sind. Deshalb können diese Anweisungen in einer Prozedur mit Wertübergabe zusammengefaßt werden, die den Namen FehlerMeldung tragen soll (siehe Seite 237).

```
type str12 = string (.12.);


procedure FehlerMeldung (Fehlerdatei : str12);
var Weiter    : char;
    BsMaske   : text;
    BildZeile : string(.80.);
begin
  assign (BsMaske,Fehlerdatei);
  reset (BsMaske);
  window (5,18,70,23);
  while not eof (BsMaske) do
  begin
    readln (BsMaske, BildZeile);
    writeln (BildZeile);
  end;
  Weiter := Readkey; (* 4 *)
(* read (kbd,Weiter); * 3 *)
  ClrScr; window (1,1,80,25);
  close (BsMaske);
end;
```

Die Prozedur FehlerMeldung wird entweder über

FehlerMeldung (Fehler01.txt)

oder über

FehlerMeldung (Fehler02.txt).

aufgerufen. Bei der Bearbeitung der Prozedur wird ein Fenster eingerichtet, das in der 5. Spalte der 18. Zeile beginnt und bis zu 70. Spalte in Zeile 23 reicht. In dieses Fenster wird der jeweilige Fehlertext ausgegeben.
Mit der Anweisung

Weiter := Readkey; (*4*)
bzw.
read (kbd,Weiter); (*3*)

wartet das Programm auf eine Tasteneingabe. Anschließend wird der Bildschirminhalt im Fenster gelöscht und wieder auf die gesamte Bildschirmgröße zurückgeschaltet.
Der übrige Teil des Programms lautet wie folgt:

```
program Zahleingabe;
uses CRT;                (* 4 *)
type str12          = string (.12.);
var  Zahl, Fehler : integer;

  procedure FehlerMeldung (Fehlerdatei : str12);
  (* siehe oben *)


begin                                        (* HAUPTPROGRAMM *)
  ClrScr;
  repeat
    write ('Eingabe einer ganzen Zahl zwischen 1 und 12');
    Fehler := 0;
    (*$I-*)
    ClrEol; readln (Zahl);
    Fehler := IOResult;
    (*$I+*)
    if Fehler <> 0
      then FehlerMeldung ('Fehler01.txt');
      else if not (Zahl in (.1..12.)
             then begin
               Fehler := 1;
               FehlerMeldung('Fehler02.txt');
             end;
  until Fehler = 0;
end.
```

In diesem Programm wird geprüft, ob der eingegebene Wert das richtige Format aufweist und ob er innerhalb des zulässigen Bereichs liegt. Geben Sie das Programm ein und testen Sie es!

7.5 Zusammenfassung

- Eine **Datei vom Typ Text** besteht aus einer beliebigen Anzahl von Zeilen. Die Zeilen dürfen unterschiedlich lang sein. Eine Zeile wird durch eine **EOL**-Marke abgeschlossen. Das Ende der Textdatei wird durch eine **EOF**-Marke gekennzeichnet.
- Eine Textdatei wird vereinbart durch
  var Dateivariable: text;

- Die Zuordnung der Dateivariablen zur physischen Datei auf einem externen Speicher geschieht durch
  assign (Dateivariable, Name_physische_Datei);
- Eine Textdatei wird mittels **rewrite (Dateivariable)** zum Schreiben geöffnet.
- Eine Textdatei wird mittels **reset (Dateivariable)** zum Lesen geöffnet.
- Eine Textdatei kann nur **zeilenweise** gelesen werden durch **readln (Dateivariable, Stringvariable);**
- Eine Datei vom Typ Text kann mittels eines eigens erstellten Programms oder über den Editor des Pascal-Systems aufgebaut werden.
- Mit der Prozedur
  window (von_Spalte,von_Zeile,bis_Spalte,bis_Zeile)
  wird ein Bildschirmausschnitt (Fenster) geschaffen.

7.6 Übungen

1. Aufgabe
Eine Datei wurde durch

```
TextA : file of string(.80.)
```

vereinbart. Wodurch unterscheidet sich die Datei TextA von einer Datei vom Typ Text?

2. Aufgabe
Wodurch wird das Ende einer Zeile in einer Textdatei gekennzeichnet?

3. Aufgabe
Welche Bedeutung hat die EOF-Marke?

4. Aufgabe
Auf welche Weise kann in Turbo-Pascal am leichtesten eine Textdatei erzeugt werden?

5. Aufgabe

Erstellen Sie mit Hilfe des Editors eine Textdatei mit einem beliebigen, mehrzeiligen Text! Speichern Sie ihn unter dem Namen DEMO16.TXT. Der Text soll in drei Fenstern auf dem Bildschirm angezeigt werden. Die Koordinaten der Fenster sind:

	Fenster 1	Fenster 2	Fenster 3
x1: von_Spalte	2	4	6
y1: von_Zeile	2	10	18
x2: bis_Spalte	78	75	72
y2: bis_Zeile	6	15	23

6. Aufgabe

In dem Programm Lagerverwaltung1 (siehe Kapitel 5.3.7, Seite 150) soll die Prozedur Buchen (siehe Kapitel 5.3.4, Seite 140) in folgender Weise ergänzt werden:

1. Bei Eingabe von Artikelnummer in B_ArtNr ist zu prüfen, ob es sich um eine positive ganze Zahl handelt.
   Wurde ein unzulässiger Wert eingegeben, so ist in einem Fenster mit den Koordinaten (1,18) und (80,23) folgender Text auszugeben:

```
Es dürfen nur positive ganze Zahlen eingegeben werden.
Drücken Sie die <RETURN>-Taste und wiederholen Sie die Eingabe!
```

2. Wird in die Variable MengenAenderung ein negativer Wert eingegeben, dessen Betrag größer ist als der Bestand, so soll die folgende Meldung in einem Fenster (Koordinaten wie oben) angezeigt werden:

```
Eingabe nicht zulässig, da die abzubuchende Menge größer ist als der
Lagerbestand.
Drücken sie die <RETURN>-Taste und wiederholen Sie die Eingabe!
```

3. Wenn eine eingegebene Artikelnummer nicht existiert, ist die Fehlermeldung

```
Diese Artikelnummer gibt es nicht.
Drücken Sie die <RETURN>-Taste und wiederholen Sie die Eingabe!
```

in einem Fenster anzuzeigen (Koordinaten des Fensters wie oben).

Erfassen Sie mit Hilfe des Editors den Text der ersten Fehlermeldung und speichern Sie ihn in der Datei mit dem Namen Hilfe01.txt! Verfahren Sie entsprechend mit dem zweiten und dritten Fehlertext! Speichern Sie sie unter dem Namen Hilfe02.txt sowie Hilfe03.txt! Entwickeln Sie die Prozeduren für die Eingabe von B_ArtNr und MengenAenderung sowie für das Einlesen und Anzeigen der Fehlermeldungen!

7.7 Lösungen

1. Aufgabe
Die Datei TextA ist eine satzorientierte Datei. Jeder Dateneintrag hat dieselbe Länge. Somit kann über die Seek-Prozedur der Satzzeiger auf einen bestimmten Datensatz gestellt werden. Bei einer Datei vom Typ Text dagegen können die einzelnen Komponenten (Zeilen) unterschiedliche Längen aufweisen. Deshalb kann eine bestimmte Komponente innerhalb der Datei nicht direkt adressiert werden. Die Datei kann nur sequentiell beschrieben oder gelesen werden.

2. Aufgabe
Das Ende einer Zeile in einer Textdatei wird durch eine EOL-Marke angezeigt.

3. Aufgabe
Die EOF-Marke kennzeichnet das Ende der Datei.

4. Aufgabe

Eine Textdatei kann schnell und problemlos mit Hilfe des Editors des Pascal-Systems erstellt werden.

5. Aufgabe

```
program Zahleingabe;
uses CRT;          (* 4 *)
var BsMaske   : text;
    BildZeile : string(.80.);

procedure Dateilesen;
begin
  assign (BsMaske,'demo16.txt');
  reset (BsMaske);
  while not eof (BsMaske) do
  begin
    readln (BsMaske, BildZeile);
    writeln (BildZeile);
  end;
  close (BsMaske);
end;

begin
  window (2,2,78,6);
  Dateilesen;
  window (4,10,75,15);
  Dateilesen;
  window (6,10,72,23);
  Dateilesen;
end.
```

6. Aufgabe

```
type str12 = string (.12.);


procedure Fehlermeldung (Fehlerdatei : str12);
var Weiter    : char;
    Bildzeile : string (.80.);
    BsMaske   : text;
begin
  assign (BsMaske, Fehlerdatei);
  reset (BsMaske);
  window (1,18,80,23);
  while not eof (BsMaske) do
  begin
    readln (BsMaske, Bildzeile);
    writeln (Bildzeile);
  end;
  Weiter := ReadKey; (*3: read (kbd, Weiter); *)
  ClrScr; window (1,1,80,25);
  close (BsMaske);
end;


procedure EinMengenAenderung (var EinMenge : integer;
                              ArtMenge : integer);
var richtig : boolean;
begin
  repeat
    richtig := true;
    Gotoxy (21,11); ClrEol; readln (EinMenge);
    if (EinMenge < 0) and (abs(EinMenge) > ArtMenge)
      then begin
        Fehlermeldung ('Hilfe02.txt');
        richtig := false;
      end;
  until richtig;
end;


procedure BArtNrEingabe (var ArtNr : integer);
var Fehler : integer;
begin
  repeat
    (*$I-*)
    Gotoxy (21,3); ClrEol; readln (ArtNr);
    Fehler := IOResult;
    if (Fehler <> 0) or (ArtNr < 0)
      then Fehlermeldung ('Hilfe01.txt');
  until Fehler = 0;
  (*$I+*)
end;
```

```
begin                                          (* Buchen *)
  B_Bildschirm;
  repeat
    ArtNrEingabe (B_ArtNr);
    Suchen_Datensatz (B_LagDatei, B_Artikel, B_ArtNr,
                      B_SatzNr,B_Gefunden);
    if B_Gefunden
      then begin
        with B_Artikel do
        begin
          Gotoxy (21,5); ClrEol;writeln (Bezeichnung);
          Gotoxy (21,7); ClrEol;writeln (Menge);
          Gotoxy (21,9); ClrEol;writeln (Preis:6:2);
          EinMengenAenderung (MengenAenderung, Menge);
          Menge := Menge + MengenAenderung;
          Gotoxy (21,7); ClrEol;writeln (Menge);
        end; (* with*)
        seek (B_LagDatei, B_SatzNr - 1);
        write (B_LagDatei, B_Artikel);
      end
      else FehlerMeldung ('Hilfe03.txt');
    Gotoxy (5,13); write ('Weiter (j/n) ');
    Weiter := ReadKey; (* 3:  read (kbd,Weiter); *)
    DelLine;
  until Weiter in ['n','N'];
end;
```

Tausendmal schneller . . .

... als herkömmliche Geräte sollen die Computer der «fünften Generation» rechnen können, die in Japan bis zum Ende dieses Jahrzehnts entwickelt werden. Sie sollen sehen, hören und sprechen können – «intelligent» sein. Aber selbst ein billiger Taschenrechner aus dem Supermarkt ist heute schon intelligent genug, die günstigste Geldanlage zu ermitteln – wenn er nur Zins und Zinseszins errechnen kann.

Pfandbrief und Kommunalobligation

Meistgekaufte deutsche Wertpapiere - hoher Zinsertrag - bei allen Banken und Sparkassen

Verbriefte **Sicherheit**

8 DYNAMISCHER DATENTYP ZEIGER (POINTER)

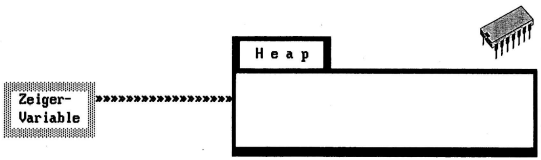
Einführungsbeispiel:
Viele Problemstellungen sind mittels der uns bekannten Datenstrukturen ARRAY oder FILE nicht oder nicht optimal lösbar. Nehmen wir einmal an:

Auf einem großen Sportfest sollen die Teilnehmerliste und der Ergebnisdienst per Computer geführt werden. In dem zu erstellenden Sportverwaltungsprogramm wird eine äußerst hohe Verarbeitungsgeschwindigkeit verlangt. Ferner ist zu berücksichtigen, daß die Anzahl der Teilnehmer zum Zeitpunkt der Programmerstellung nicht bekannt ist. Wie viele Sportler an dem Sportfest teilnehmen, steht vielmehr erst kurz vor Beginn fest.

In welcher Datenstruktur sind die Teilnehmerdaten zu speichern? Eine ARRAY-Struktur, die man zunächst wohl in Erwägung ziehen könnte, ist nicht brauchbar. Bei ihr muß bereits bei der Erstellung des Programms festgelegt werden, wie groß der ARRAY sein soll. Das widerspricht jedoch der Einsatzbedingung des Programms.
Es wird somit eine Datenstruktur benötigt, die im Arbeitsspeicher aufgebaut wird und deren Umfang während des Programmlaufs veränderbar ist. Eine solche Datenstruktur wird **dynamische Datenstruktur** genannt. Ein ARRAY ist dagegen eine **statische Datenstruktur**. Ihr Umfang kann während des Programmlaufs nicht verkleinert oder vergrößert werden.
Zum Aufbau von dynamischen Datenstrukturen wird der Datentyp **Zeiger** benötigt. Er wird oftmals auch als **POINTER** (engl. Zeiger) bezeichnet. Vor der Entwicklung des Sportverwaltungsprogramms wird deshalb der Datentyp Zeiger erläutert.

8.1 Zeigervariable

Eine dynamische Datenstruktur baut sich im freien Teil des Arbeitsspeichers auf. Er wird **Heap** (engl. Haufen) genannt. Der Zugriff auf den Heap erfolgt über eine Variable, die als **Zeigervariable** bezeichnet wird.



8.1.1 Vereinbarung und Zugriff auf eine dynamische Variable

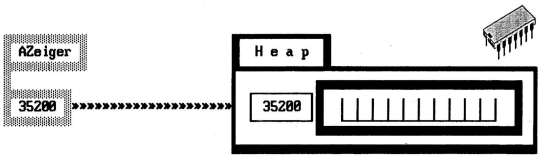
Im folgenden Programmbeispiel wird die Vereinbarung einer Zeigervariable dargelegt. Ferner wird aufgezeigt, wie eine dynamische Variable erzeugt und wie auf sie zugegriffen wird.

```
1  program zeiger1;
2  uses CRT;      (* 4 *)
3  type string25 = string (.25.);
4  var  AZeiger   : ^string25;
5  begin
6    ClrScr;
7    new (AZeiger);
8    write ('Eingabe Name: '); readln (AZeiger^);
9    writeln;
10   writeln (AZeiger^);
11 end.
```

Vereinbarung einer Zeigervariablen
In Zeile 4 wird eine Zeigervariable mit dem Namen AZeiger vereinbart. Die Vereinbarung ^**string25** gibt an, daß die Variable AZeiger auf eine dynamische Variable zeigen soll, deren Datentyp string25 ist. (In Standard-Pascal wird statt ^ der Hochpfeil verwendet).

Das Symbol ^ kennzeichnet die Variable als Zeigervariable. Hinter dem Zeiger-Symbol wird angegeben, von welchem Datentyp die Variable ist, auf die die Zeigervariable zeigen soll.
Die dynamische Variable selber wird nicht im Vereinbarungsteil angegeben. Es wird lediglich die Zeigervariable, die auf die dynamische Variable zeigen soll, vereinbart.

Erzeugen einer dynamischen Variablen
Eine dynamische Variable wird erst während des Programmlaufs durch die Standardprozedur **NEW** eingerichtet. Mit der Prozedur **NEW (AZeiger)** in Zeile 7 wird auf dem Heap der Speicherplatz für eine 25 Stellen lange Zeichenkette eingerichtet. In der Zeigervariablen wird die Speicheradresse dieses Speicherplatzes vermerkt.



Zugriff auf eine dynamische Variable
Den Zugriff auf eine dynamische Variable verdeutlicht die Anweisung **readln (AZeiger^)** (sprich: AZeiger Hochpfeil). Die eingegebene Zeichenkette wird ab dem Speicherplatz gespeichert, dessen Adresse in der Zeigervariablen AZeiger steht. Anders ausgedrückt: Die Zeichenkette wird auf dem Speicherplatz gespeichert, auf den die Zeigervariable zeigt. Entsprechend wird mit **writeln (AZeiger^)** der Inhalt des Speicherplatzes ausgegeben, auf den die Variable AZeiger verweist.
Eine dynamische Variable hat keinen eigenen Namen, über den auf sie zugegriffen werden könnte. Man kann sich auf sie nur über eine Zeigervariable beziehen, z. B. über AZeiger^. Deshalb bezeichnet man die dynamische Variable AZeiger^ auch als **Bezugsvariable** von AZeiger.

8.1.2 Speicherplatzfreigabe

Eine dynamische Variable kann auch während des Programmlaufs wieder "gelöscht" werden. Unter Löschen wird hier verstanden, daß der für die dynamische Variable reservierte Speicherplatz wieder freigegeben wird. Dazu wird die Prozedur **Dispose** (engl. beseitigen) verwendet. Die Wirkung von Dispose zeigt das nachstehende Programm.

```
1  program Zeiger1b;
2  uses CRT;          (* 4 *)
3  type string25 = string (.25.);
4  var AZeiger   : ^string25;
5  begin
6    ClrScr;
7    writeln  ('Freier heap ',memavail);
8    new (AZeiger);
9    write ('Eingabe Name: '); readln (AZeiger^);
10   writeln;
11   writeln ('Freier heap nach New ',memavail);
12   dispose (AZeiger);
13   writeln ('Freier heap nach Dispose ',memavail);
14 end.
```

Die Größe des freien Speicherplatzes auf dem Heap wird über die Systemvariable **Memavail** (= Memory available = verfügbarer Speicher) angezeigt. Wenn Sie das obige Programm eingeben, starten und irgendeinen Namen eingeben, erscheint folgende Ausgabe am Bildschirm:

```
Freier heap 361248
Eingabe Name: Turbo-Pascal

Freier heap nach New 361222
Freier heap nach Dispose 361248
```

Zu Beginn des Programms beträgt der freie Speicherplatz auf dem Heap bei 640 KB Arbeitsspeicher 361 248 Bytes. Nachdem durch

```
new (Zeiger)
```

ein Speicherplatz für eine 25stellige Zeichenkette zuzüglich eines Längenbytes reserviert worden ist, beträgt der freie Speicher auf dem Heap nur noch 361 222 Bytes. Durch **DISPOSE (AZeiger)** wird der Speicherplatz für die Bezugsvariable AZeiger^ freigegeben. Es sind dann wieder 361 248 Bytes verfügbar. Bei der nächsten NEW-Anweisung kann dieser Speicherbereich erneut genutzt werden.
Wenn noch weitere dynamische Variablen auf dem Heap geführt werden, so werden sie durch den DISPOSE-Befehl nicht berührt.

Durch **new (Zeiger)** wird auf dem Heap Speicherplatz entsprechend der Größe der Bezugsvariablen reserviert. Durch **Dispose (Zeiger)** wird der Speicherplatz wieder freigegeben.

8.2 Operationen auf Zeigervariablen

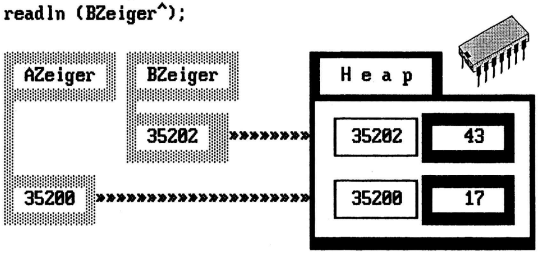
Auf Zeigervariablen dürfen **nur Wertzuweisungen** und **Vergleichsoperationen** angewendet werden. Ein- und Ausgabeoperationen sind also nicht möglich.

Wertzuweisung
Durch eine Wertzuweisung kann der Wert einer Zeigervariablen einer anderen zugewiesen werden. So wird beispielsweise durch

```
BZeiger := AZeiger;
```

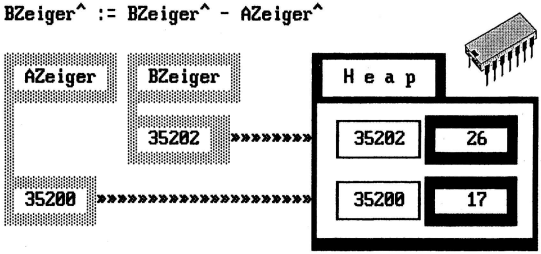
die Adresse, die in der Zeigervariablen AZeiger steht, in die Zeigervariablen BZeiger kopiert. Beide Zeiger zeigen somit auf dieselbe dynamische Variable. Aber Vorsicht: Wenn BZeiger vor der Wertzuweisung die Heap-Adresse einer dynamischen Variablen beinhaltete, so ist diese dynamische Variable nach der Wertzuweisung nicht mehr erreichbar.
Voraussetzung für die Wertzuweisung zwischen Zeigervariablen ist, daß ihre Bezugsvariablen **denselben Datentyp** haben.
Den Zusammenhang zwischen Wertzuweisungen auf Zeigervariablen und Bezugsvariablen verdeutlicht folgende Abbildung auf Seite 250:

```
readln (AZeiger^);
readln (BZeiger^);
```



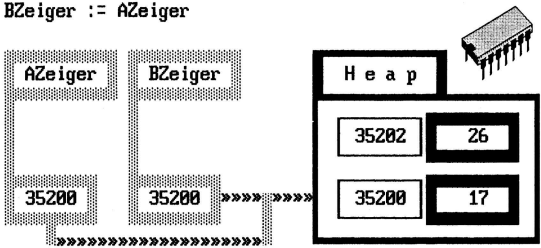
Subtraktion der Bezugsvariablen

```
BZeiger^ := BZeiger^ - AZeiger^
```



Die Wertzuweisung

BZeiger := AZeiger



bewirkt, daß BZeiger nun ebenfalls den Adreßwert 35200 aufweist. Ein Zugriff auf die gerade gebildete Differenz ist nicht mehr möglich. BZeiger zeigt jetzt auf die Bezugsvariable von AZeiger.

Vergleichsoperationen

In Vergleichsoperationen kann geprüft werden, ob zwei Zeigervariablen dieselbe Heap-Adresse beinhalten. Ein Vergleich zwischen zwei Zeigervariablen ist nur zulässig, wenn ihre Bezugsvariablen denselben Datentyp haben.

Beispiel:

```
if AZeiger <> BZeiger
then write ('Die Zeiger zeigen auf unterschiedliche Objekte')
else write ('Die Zeiger zeigen auf dasselbe Objekt');
```

Durch diese Abfrage wird also geprüft, ob die beiden Zeiger dieselbe Bezugsvariable haben oder auf unterschiedliche Bezugsvariablen zeigen.

NIL

Eine besondere Bedeutung hat der spezielle Zeigerwert **NIL** (lat. **Nihil** = nichts). Er besagt, daß der Zeiger auf nichts zeigt. Dieser Wert darf jeder Zeigervariablen zugewiesen werden. Auf diesen Wert darf auch jede Zeigervariable verglichen werden, weil NIL zu allen Datentypen kompatibel ist. Die Wertzuweisung

```
AZeiger := NIL
```

bewirkt, daß die Zeigervariable AZeiger auf nichts zeigt. Durch

```
if AZeiger = NIL
then write ('Der Zeiger zeigt auf nichts');
```

wird geprüft, ob AZeiger auf nichts zeigt.
NIL ist ein reserviertes Wort.
Die Wirkung von NIL können Sie selber ausprobieren, indem Sie in dem Programm Zeiger1 vor die Ausgabeanweisung 10 writeln (AZeiger^) die Wertzuweisung

```
AZeiger := NIL;
```

setzen und das Programm starten. Nach der Eingabe eines Namens erscheint möglicherweise eine unsinnige Ausgabe auf dem Bildschirm, weil der Zeiger auf "nichts" zeigt.

8.3 Zusammenfassung

- Eine dynamische Variable ist dadurch gekennzeichnet, daß sie erst während des Programmlaufs eingerichtet wird. Sie kann auch während des Programmlaufs wieder aufgehoben werden.
- Eine dynamische Variable wird auf dem Heap aufgebaut.
- Auf eine dynamische Variable kann nur mittels Zeiger zugegriffen werden.
- Ein Zeiger wird durch

  ```
  var Zeigervariable: ^Datentyp;
  ```

  vereinbart.
- Mittels **NEW (Zeigervariable)** wird Speicherplatz für die Bezugsvariable auf dem Heap reserviert. In der Zeigervariablen steht die Anfangsadresse des reservierten Speicherbereichs.
- Der Zugriff auf die Bezugsvariable erfolgt durch Zeigervariable^.
- Durch **DISPOSE (Zeigervariable)** wird der Speicherplatz auf dem Heap wieder freigegeben.
- Auf Zeigervariablen sind **nur Wertzuweisungen** und **Vergleichsoperationen** zulässig. Dabei müssen die zugehörigen Bezugsvariablen denselben Datentyp haben.
- Der Zeigerwert **NIL** bedeutet, daß der Zeiger auf nichts zeigt. NIL ist zu allen Datentypen kompatibel.

8.4 Übungen

1. Aufgabe
Welcher Unterschied besteht zwischen einer statischen und einer dynamischen Variablen?

2. Aufgabe
Es ist eine Zeigervariable BZeiger zu vereinbaren, deren Bezugsvariable vom Typ Real ist. Wie lautet die Vereinbarung?

3. Aufgabe
Was bewirkt die Anweisung new (BZeiger)?

4. Aufgabe
In die Bezugsvariable, auf die BZeiger zeigt, ist ein Wert einzugeben. Wie lautet die Read-Anweisung?

5. Aufgabe
Gegeben sei folgendes Programm:

```
program ZeigerUebung1;
var Azeiger, Bzeiger, Czeiger: ^integer;
begin
  new (Azeiger); new (Bzeiger); new (Czeiger);
  Azeiger^ := 10;
  Bzeiger^ := 5;
  _____
  _____
  end.
```

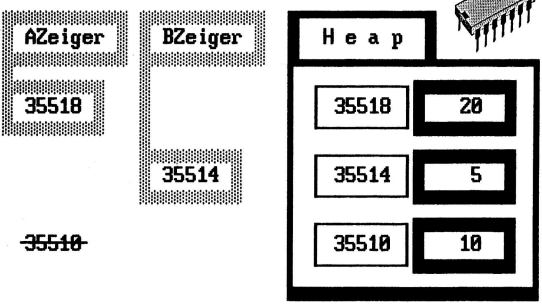
In der Bezugsvariablen zu Czeiger ist das Produkt der Bezugsvariablen von Azeiger und Bzeiger zu bilden und auszugeben. Ergänzen Sie das Programm um die beiden fehlenden Anweisungen!

6. Aufgabe
Welche Wirkung hat die Anweisung DISPOSE (AZeiger)?

7. Aufgabe
Gegeben sei folgendes Programm:

```
program ZeigerUebung2;
var Azeiger, Bzeiger : ^integer;
    Produkt          : integer;
begin
  new (Azeiger); new (Bzeiger);
  Azeiger^ := 10;
  Bzeiger^ :=  5;
  new (Azeiger);
  Azeiger^ := 20;
  Produkt  := Azeiger^ * Bzeiger^;
end.
```

Wertbelegungstabelle



a) Mit welchen Werten wird die Multiplikation vorgenommen?
b) Warum kann auf den Wert 10, der ab der Heap-Adresse 35510 gespeichert ist, nicht mehr zugegriffen werden?

8.5 Lösungen

1. Aufgabe
Eine dynamische Variable wird im Gegensatz zur statischen Variablen nicht im Vereinbarungsteil vereinbart. Sie wird erst während des Programmlaufs eingerichtet und kann auch während des Programms wieder aufgelöst werden.

2. Aufgabe
Die Zeigervereinbarung lautet:

```
var BZeiger: ^real;
```

3. Aufgabe
Durch new (BZeiger) wird Speicherplatz zur Aufnahme eines Real-Wertes reserviert.

4. Aufgabe
Die Lese-Anweisung lautet:

```
readln (BZeiger^);
```

5. Augabe
Das Programm ist zu ergänzen um

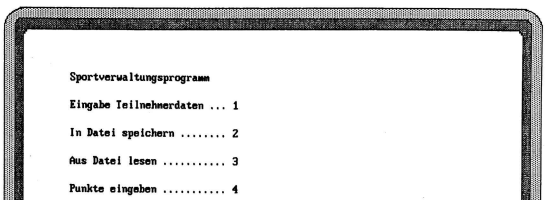
```
Czeiger^: = Azeiger^ * Bzeiger^;
writeln ('Das Produkt lautet:', Czeiger^: 10);
```

6. Aufgabe
Durch DISPOSE (Azeiger) wird der Speicherplatz, auf den Azeiger zeigt, freigegeben. Die Bezugsvariable wird "gelöscht".

7. Aufgabe
a) Die Multiplikation wird mit den Werten 20 und 5 vorgenommen.
b) Ein Zugriff ist nicht mehr möglich, da kein Zeiger mehr auf den Speicherplatz zeigt. Der Speicherplatz ist aber noch belegt, weil er noch nicht durch DISPOSE freigegeben wurde.

9 DYNAMISCHE DATENSTRUKTUR: VERKETTETE LISTE

Einführungsbeispiel:
Für die Organisation einer großen Sportveranstaltung wird ein Programm benötigt, das mit maximaler Arbeitsgeschwindigkeit die Teilnehmerdaten verarbeitet. Die Anzahl der Sportler steht erst unmittelbar vor Beginn der Veranstaltung fest.
Der Leistungsumfang des zu entwickelnden Programms ist aus der nachstehenden Abbildung des Auswahlbildschirms zu ersehen.



Die einzelnen Programmfunktionen werden in diesem Kapitel schrittweise entwickelt.
Der Datensatz eines Teilnehmers soll aus einem Record bestehen:

```
type Teilnehmersatz = record
                        Nr    : integer;
                        Name  : str25
                        Punkte : integer;
                      end;
```

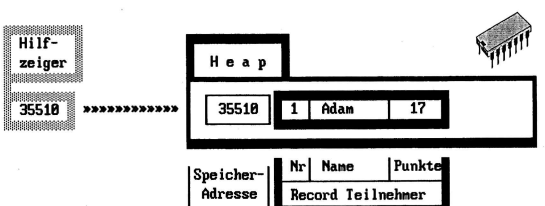
9.1 Grundstrukturen

9.1.1 Charakteristik einer verketteten Liste

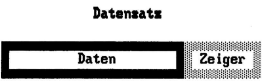
Die verkettete Liste basiert auf den Grundlagen der Zeigertechnik, wie sie im Kapitel 8 dargelegt wurde. Um die Daten eines Teilnehmers auf dem Heap speichern zu können, muß eine Zeigervariable vereinbart werden, deren Bezugsvariable vom Typ Teilnehmersatz ist:

```
var Zeiger: ^Teilnehmersatz;
```

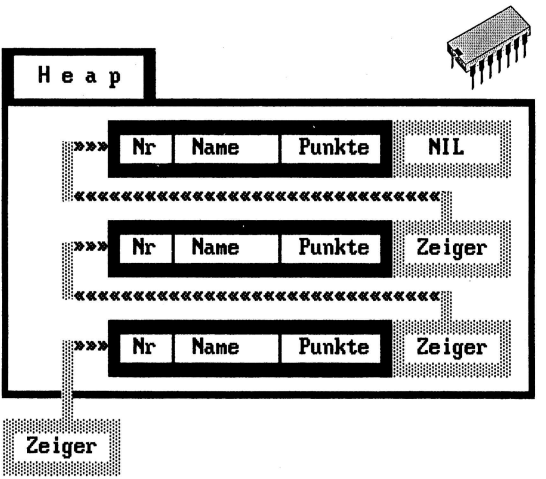
Durch new (Zeiger) wird auf dem Heap ein Speicherbereich eingerichtet, in den die Daten eines Teilnehmers eingegeben werden können.



Um die Daten des zweiten und der nachfolgenden Starter speichern zu können, müßten weitere Zeiger vereinbart und weitere Bezugsvariablen eingerichtet werden. Dies erreicht man dadurch, daß der Datensatz eines Starters um einen Zeiger erweitert wird.



Der Datensatz eines Teilnehmers besteht nunmehr aus einem **Datenteil** und einem **Zeigerteil**. In den Zeigerteil des ersten Datensatzes wird die Heap-Adresse des zweiten Teilnehmers eingetragen. Im Datensatz des zweiten Starters wird die Heap-Adresse des dritten Teilnehmers vermerkt. Beim letzten Teilnehmer schließlich weist der Zeiger auf **NIL**.



Wie aus der Abbildung zu entnehmen ist, werden die Daten nacheinander von unten nach oben im Heap aufgeschichtet. Der erste Teilnehmer steht im Haufen an unterster Stelle, der letzte an oberster Stelle.
Damit ist der prinzipielle Aufbau einer dynamischen Datenstruktur,

die **verkettete Liste** genannt wird, skizziert. Der Verkettung der einzelnen Datensätze, aus denen die Liste besteht, wird durch Zeiger erreicht. Der Zeiger des ersten Datensatzes zeigt auf den Beginn des zweiten Datensatzes, der Zeiger des zweiten Datensatzes weist auf den Beginn des dritten Datensatzes usw. Das Ende der Kette ist erreicht, wenn der Zeiger auf nichts zeigt, also den Wert NIL beinhaltet.

9.1.2 Vereinbarung der Datenstruktur

Wie bereits oben erwähnt, besteht die Bezugsvariable aus einem Datenteil und einem Zeigerteil. Der Datenteil wird in unserem Beispiel Starter genannt und ist vom Typ Teilnehmersatz. Wie der Zeigerteil vereinbart wird, ist nachstehend dargelegt.

```
type Zeiger     = ^StartListe;
     StartListe = record
                    Starter   : Teilnehmersatz;
                    Naechster : Zeiger;
                  end;
```

Es wird zunächst der Datentyp Zeiger vereinbart, der auf einen Datensatz vom Typ StartListe zeigen soll. Wie der Datentyp StartListe beschaffen ist, wird anschließend festgelegt. StartListe ist ein Verbund, der sich aus einem Datenteil und einem Zeigerteil zusammensetzt. Den Datenteil bildet Starter vom Typ Teilnehmersatz. Der Zeigerteil besteht aus dem Element Naechster. Naechster ist seinerseits vom Typ Zeiger. Die Zeigervariable Naechster zeigt also ebenfalls auf eine Bezugsvariable vom Typ StartListe.

Allgemein ist die Typvereinbarung für die dynamische Datenstruktur zum Aufbau einer verketteten Liste wie folgt aufgebaut:

```
type Zeiger =^ Datensatz;
     Datensatz = record
                   Datenteil : Datentyp;
                   Zeigerteil : Zeiger;
                 end
```

Die Typdefinition zur Konstruktion einer verketteten Liste weicht von der üblichen Form der Typdefinition ab. Normalerweise muß ein Datentyp erst definiert worden sein, bevor sein Bezeichner in einer anderen Typdefinition verwendet werden darf. Bei der Vereinbarung eines Zeigertyps wird der Bezeichner des Datentyps bereits vorher genannt. Erst anschließend wird erklärt, welche Datenstruktur mit dem Bezeichner gemeint ist.
Der so geschaffene Zeigertyp wird zur Vereinbarung der Zeigervariablen verwendet. In unserem Beispiel soll die Zeigervariable, die die Adresse des ersten Datensatzes auf dem Heap aufnimmt, Anfangszeiger genannt werden. Die Vereinbarung der Zeigervariablen lautet dann:

```
var Anfangszeiger: Zeiger;
```

Diese Zeigervariable ist von großer Bedeutung, da über diesen Zeiger stets der "Einstieg" in die Liste erfolgt.

9.1.3 Algorithmischer Ansatz zum Aufbau einer verketteten Liste

Im folgenden wird schrittweise der Algorithmus, über den eine Verkettung der Datensätze auf dem Heap erreicht wird, entwickelt. Er ist Grundlage für die Programmierung im Kapitel 9.2.
Beim Aufbau des Algorithmus müssen zwei Bedingungen beachtet werden:

1. Die Anfangsadresse des ersten Datensatzes muß in einem Zeiger festgehalten werden.
2. Der Zeiger des jeweils zuletzt eingetragenen Datensatzes muß den Wert NIL aufweisen.

Dadurch wird gewährleistet, daß man stets auf den ersten Datensatz, der auf dem Heap gespeichert ist, zugreifen kann. Mit Hilfe der Zeiger, die Bestandteil eines jeden Datensatzes auf dem Heap sind, kann man dann nacheinander die einzelnen Datensätze durchlaufen, bis man schließlich zu dem Datensatz gelangt, dessen Zeiger den Wert NIL aufweist. Für die Programmlogik hat das NIL-Merkmal dieselbe Bedeutung wie das EOF-Merkmal bei sequentiellen Dateien.

Eintrag des ersten Datensatzes

Die Zeigervariable, die auf den ersten Datensatz zeigen soll, wird hier Anfangszeiger genannt. Folgende Schritte sind für den Eintrag des ersten Datensatzes auf dem Heap erforderlich:

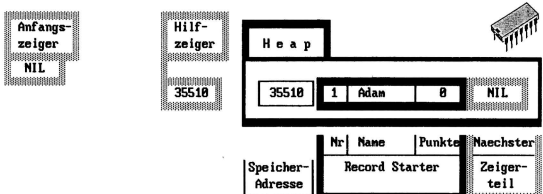
1. Anfangszeiger wird mit dem Wert NIL belegt, da der Zeiger zu Beginn auf nichts zeigen soll.

   Anfangszeiger: = NIL;

2. Mit einem weiteren Zeiger, der den Namen Hilfzeiger trägt, wird der Speicherplatz für einen Teilnehmerdatensatz auf dem Heap reserviert.

   new (Hilfzeiger);

3. In den Datenteil der so eingerichteten Bezugsvariablen werden die Daten eingetragen. In den Zeigerteil wird der Wert NIL eingetragen, da dieser Satz der zur Zeit letzte in der Liste ist.



   Die Adresse des ersten Datensatzes auf dem Heap sei 35510. Sie ist in der Zeigervariablen Hilfzeiger vermerkt.

4. Durch die Wertzuweisung

   Anfangszeiger: = Hilfzeiger;

   wird die Anfangsadresse in die Zeigervariable Anfangszeiger eingetragen.

Mit der Schrittfolge

Anfangszeiger: = NIL;
new (Hilfzeiger);
Teilnehmerdaten in Datenteil eintragen
Zeigerteil: = NIL;
Anfangszeiger: = Hilfzeiger

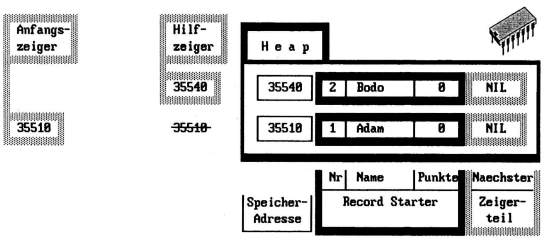
sind die beiden obigen Bedingungen erfüllt. In der Zeigervariablen Anfangszeiger befindet sich die Anfangsadresse. Der jeweils letzte Datensatz auf dem Heap beinhaltet im Zeigerteil den Wert NIL.

Eintrag von Folgesätzen

Vor der Speicherung des nächsten Datensatzes auf dem Heap wird mit

new (Hilfzeiger)

erneut Speicherplatz auf dem Heap reserviert, in den anschließend der Teilnehmerdatensatz eingetragen wird. Ferner wird in dem Zeigerteil der Wert NIL vermerkt.



Zur Verkettung der Sätze muß die Adresse des gerade gespeicherten Datensatzes in den Zeigerteil des vorhergehenden Datensatzes eingetragen werden. Dieser vorhergehende Satz ist dadurch gekennzeichnet, daß in seinem Zeigerteil der Wert NIL steht. Um diesen Satz zu finden, wird die Liste vom Anfang an nach dem Satz durchsucht, dessen Zeigerteil den Wert NIL aufweist.
Zum Durchsuchen der bisher gespeicherten Datensätze wird noch ein

dritter Zeiger mit dem Namen Laufzeiger herangezogen, dem mittels

Laufzeiger: = Anfangszeiger

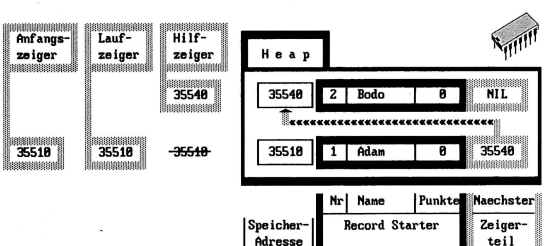
die Anfangsadresse des ersten Datensatzes zugewiesen wird. In der Suchroutine darf Anfangszeiger nicht zur Adressierung verwendet werden, weil in diesem Zeiger die "Einstiegsadresse" gespeichert ist. In einer WHILE-Schleife werden jetzt von Beginn an die Zeigerteile der Datensätze auf dem Heap nach dem NIL-Merkmal durchsucht. In unserem Beispiel wird bereits bei der ersten Abfrage des Zeigerteils das Suchmerkmal gefunden, weil im Zeigerteil des ersten Datensatzes das NIL-Merkmal steht. Daraufhin wird in den Zeigerteil der Bezugsvariablen von Laufzeiger der Inhalt von Hilfzeiger eingetragen. In Hilfzeiger steht, wie oben dargestellt, die Adresse des folgenden Datensatzes.

Zeigerteil der Bezugsvariablen von Laufzeiger := Hilfzeiger
bzw. in Pascal-Schreibweise:

```
Laufzeiger^.Naechster := Hilfzeiger;
```

Bezugsvariable von Laufzeiger
Zeigerteil der Bezugsvariablen

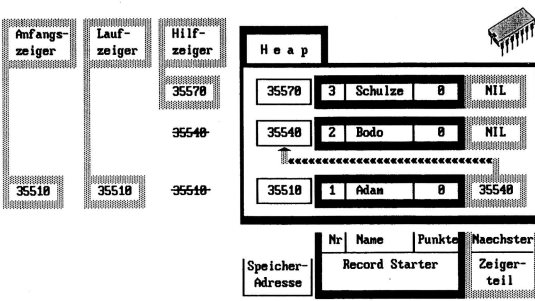
Der Zeigerteil des ersten Satzes verweist jetzt also auf den zweiten Datensatz.



Ist nun ein weiterer Datensatz einzutragen, so wird der Speicherplatz dafür wieder durch

new (Hilfzeiger)

reserviert. Die Daten des Teilnehmers werden eingetragen, und der Zeigerteil des Datensatzes wird auf NIL gesetzt.



Auch jetzt muß zur Verkettung wieder der vorhergehende, bislang letzte Datensatz gefunden werden. Dem Laufzeiger wird mittels

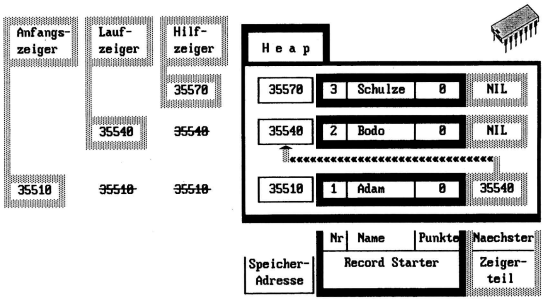
Laufzeiger: = Anfangszeiger

die Anfangsadresse des ersten Satzes zugewiesen. Solange der Zeigerteil der Bezugsvariablen von Laufzeiger nicht das NIL-Merkmal, sondern die Adresse des Folgesatzes enthält, wird dem Laufzeiger der Inhalt des Zeigerteils zugewiesen:

Laufzeiger: = Zeigerteil des Datensatzes, der durch Laufzeiger adressiert ist

bzw. in Pascal-Schreibweise:

Laufzeiger: = Laufzeiger^. Naechster;



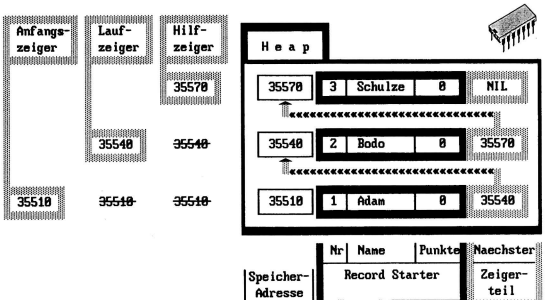
Befindet sich in dem Zeigerteil der Bezugsvariablen von Laufzeiger das NIL-Merkmal, so muß in den Zeigerteil die Adresse, die in Hilfzeiger steht, eingetragen werden:

Zeigerteil der Bezugsvariablen von Laufzeiger: = Hilfzeiger

bzw. in Pascal-Schreibweise:

Laufzeiger^. Naechster: = Hilfzeiger

Das Ergebnis dieser Wertzuweisung zeigt folgende Abbildung:



Der Zeigereintrag in den bislang letzten Datensatz erfolgt also nach folgendem Algorithmus:

```
Laufzeiger := Anfangszeiger
Solange Zeigerteil der Bezugsvariablen von Laufzeiger
ungleich NIL ist
    Laufzeiger := Zeigerteil der Bezugsvariablen von Laufzeiger
Zeigerteil der Bezugsvariablen von Laufzeiger := Hilfzeiger
```

Diese Anfügeroutine darf jedoch nur eingesetzt werden, wenn es sich um den Eintrag von Folgesätzen handelt. Beim ersten Satz darf sie nicht angewendet werden. Ob es sich um einen ersten Eintrag oder um einen Folgeeintrag handelt, ist aus dem Wert des Zeigers Anfangszeiger zu erkennen. Er weist beim ersten Eintrag zunächst den Wert NIL auf.

Algorithmus

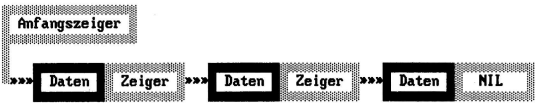
Der Algorithmus zum Anfügen und zur Verkettung neu eingetragener Datensätze lautet somit:

```
Algorithmus Anfügen von Datensätzen mit Verkettung
new (Hilfzeiger)
Datensatz auf dem Heap eintragen
Zeigerteil der Bezugsvariablen von Hilfzeiger := NIL
                Wenn Anfangszeiger = NIL
        ja                          nein
Anfangszeiger :=    Laufzeiger := Anfangszeiger
Hilfzeiger
                    Solange Zeigerteil der Bezugsvariablen von
                    Laufzeiger ungleich NIL
                        Laufzeiger := Zeigerteil der Bezugsvariablen
                                      von Laufzeiger
                    Zeigerteil der Bezugsvariablen von Laufzeiger :=
                    Hilfzeiger
```

Dieser Algorithmus bewirkt, daß in der Zeigervariablen Anfangszeiger die Anfangsadresse der dynamischen Variablen auf dem Heap vermerkt wird. Im Zeigerteil eines jeden Datensatzes auf dem Heap befindet sich die Adresse des Folgesatzes. Der letzte Satz trägt das Ende-Merkmal NIL. Der Algorithmus erfüllt also die beiden oben aufgestellten Bedingungen.

9.1.4 Zusammenfassung

- Zum Aufbau einer verketteten Liste ist es erforderlich, daß der Datensatz der Bezugsvariablen aus einem **Datenteil** und einem **Zeigerteil** besteht.
- Die Anfangsadresse des ersten Datensatzes auf dem Heap wird in einer Zeigervariablen, häufig Anfangszeiger genannt, vermerkt.
- Die Verkettung der Datensätze wird dadurch erreicht, daß der Zeigerteil eines Datensatzes auf den direkt folgenden Datensatz zeigt.
- Der letzte Datensatz einer verketteten Liste beinhaltet den Zeigerwert **NIL**.
- Schematischer Aufbau einer verketteten Liste:



- Die Vereinbarung einer Zeigervariablen zum Aufbau einer verketteten Liste erfolgt in zwei Schritten:

1. Schritt: Vereinbarung des Zeigertyps

```
type Zeiger = ^ Datensatz;
Datensatz   = record
                Datenteil: Datentyp;
                Zeigerteil: Zeiger;
              end;
```

2. Schritt: Vereinbarung der Zeigervariablen

```
var ZeigerVariable: Zeiger;
```

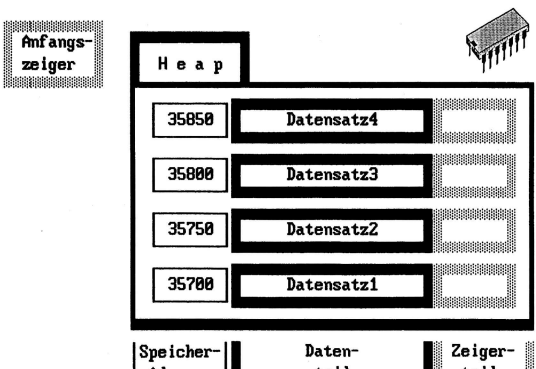
9.1.5 Übungen

1. Aufgabe
Ein Artikelsatz bestehe aus

Artikelnr.: integer
Artikelbezeichnung: string (.35.)
Menge: integer
Einzelpreis: real

Die Daten aller Artikel sollen in einer verketteten Liste dargestellt werden. Wie lautet die Vereinbarung des Zeigertyps und der Zeigervariablen?

2. Aufgabe
Gegeben ist folgender Dateneintrag auf dem Heap:



Mit welchen Werten müssen die Zeiger belegt sein, damit eine verkettete Liste entsteht?

3. Aufgabe
Warum wird beim Eintrag eines Datensatzes der Zeigerteil zunächst auf NIL gesetzt?

4. Aufgabe
Wie lautet die WHILE-Schleife zum Suchen des bislang letzten Datensatzes, wenn in der Zeigervariablen Laufzeiger die Adresse des ersten Datensatzes steht?

5. Aufgabe
In der Zeigervariablen Hilfzeiger steht die Adresse des zuletzt eingetragenen Datensatzes. Die Zeigervariable Laufzeiger beinhaltet die Adresse des bislang letzten Datensatzes. Wie lautet die Wertzuweisung, um den bislang letzten Datensatz mit dem zuletzt eingetragenen Datensatz zu verketten?

6. Aufgabe
Die Artikeldaten (siehe Aufgabe 1) sind in einer verketteten Liste darzustellen. Ergänzen Sie den folgenden Grob-Algorithmus zum Aufbau einer verketteten Liste! Vor dem ersten Dateneintrag weist die Zeigervariable Anfangszeiger den Wert NIL auf.

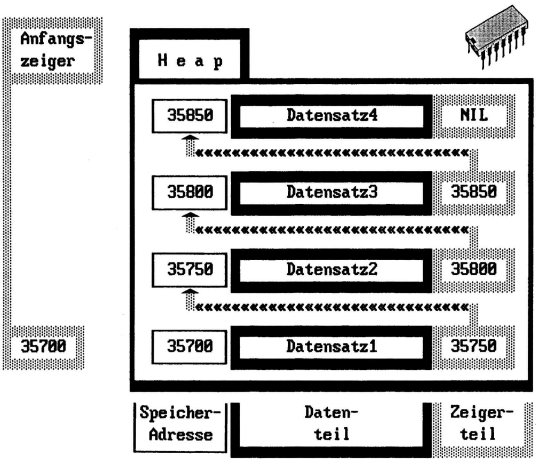
```
  ...
New (Hilfzeiger);
Hilfzeiger^.Artikel := Artikeldaten;
_____;
if _____ = NIL
 then Anfangszeiger := _____
 else begin
      Laufzeiger := _____;
      while _____ <> NIL
        do Laufzeiger := _____;
      _____;
      end;
  ...
```

9.1.6 Lösungen

1. Aufgabe

```
type Artikelsatz = record
                     Nr          : integer;
                     Bezeichnung : string (.35.);
                     Menge       : integer;
                     Einzelpreis : real;
                   end;
     Zeiger      = ^ArtSatz;
     ArtSatz     = record
                     Artikel     : Artikelsatz;
                     Naechster   : Zeiger;
                   end;
var  ArtZeiger : Zeiger;
```

2. Aufgabe



3. Aufgabe

Der jeweils letzte Datensatz einer verketteten Liste muß den Zeigerwert NIL beinhalten.

4. Aufgabe

while Zeiger der Bezugsvariablen von Laufzeiger ungleich NIL
    do Laufzeiger := Zeigerteil der Bezugsvariablen
                     von Laufzeiger

bzw.

while Laufzeiger^. Naechster <> NIL
    do Laufzeiger := Laufzeiger^. Naechster;

5. Aufgabe

Zeigerteil der Bezugsvariablen von Laufzeiger: = Hilfzeiger

bzw.

Laufzeiger^. Naechster := Hilfzeiger

6. Aufgabe

```
...

new (Hilfzeiger);
Hilfzeiger^.Artikel := Artikeldaten;
Hilfzeiger^.Naechster := Nil;
if Anfangszeiger = NIL
  then Anfangszeiger := Hilfzeiger
  else begin
    Laufzeiger := Hilfzeiger;
    while Laufzeiger^.Naechster <> NIL do
      Laufzeiger := Laufzeiger^.Naechster;
    Laufzeiger^.Naechster := Hilfzeiger;
  end;

...
```

9.2 Anwendungen

Die Verarbeitung von verketteten Listen wird an Hand der Entwicklung eines Sportverwaltungsprogramms aufgezeigt.

9.2.1 Programmrahmen

Für die Organisation der Sportveranstaltung soll das Programm über folgende Funktionen verfügen:

- Eingabe der Teilnehmerdaten mit Aufbau der verketteten Liste
- Speichern der Teilnehmerdaten in einer Datei
- Eingabe von Wettkampfergebnissen (Punkte)
- Lesen der Teilnehmerdatei mit Aufbau der verketteten Liste
- Lesen der Teilnehmerdatei mit Aufbau einer nach Punkten sortierten verketteten Liste
- Anzeige der Teilnehmerdaten am Bildschirm

Die einzelnen Funktionen werden in Prozeduren behandelt. Die Prozeduren werden aus dem Hauptprogramm aufgerufen, das zusammen mit der Prozedur Auswahlmaske nachstehend dargestellt ist.

```
program Sportverwaltung;
uses CRT;              (* 4 *)
type string25       = string (.25.);
     Teilnehmersatz = Record
                        Nr        : Integer;
                        Name      : String25;
                        Punkte    : Integer;
                      end;
     Zeiger         = ^StartListe;
     StartListe     = Record
                        Starter   : Teilnehmersatz;
                        Naechster : Zeiger;
                      end;

var  Anfangszeiger : Zeiger;
     Weiter        : char;
     Auswahl       : Integer;
```

```
procedure Auswahlmaske;
begin
  ClrScr;
  Gotoxy (5,5);  writeln ('Sportverwaltungsprogramm');
  Gotoxy (5,7);  writeln ('Eingabe Teilnehmerdaten ... 1');
  Gotoxy (5,9);  writeln ('In Datei speichern  ..... 2');
  Gotoxy (5,11); writeln ('Aus Datei lesen ..... 3');
  Gotoxy (5,13); writeln ('Punkte eingeben ..... 4');
  Gotoxy (5,15); writeln ('Aus Datei sortiert lesen .. 5');
  Gotoxy (5,17); writeln ('Teilnehmerdaten anzeigen .. 6');
  Gotoxy (5,20); writeln ('Auswahl (0 = Ende) ..... ');
end;


begin                                    (* HAUPTPROGRAMM*)
  Anfangszeiger:= NIL;
  Auswahlmaske;
  Gotoxy (33,20); ClrEol; readln (Auswahl);
  repeat
    case Auswahl of
      1  : Eingabe (Anfangszeiger);
      2  : ListeInDateiSchreiben(Anfangszeiger);
      3  : begin
             Anfangszeiger := NIL;
             Dateilesen_Listenaufbau(Anfangszeiger);
           end;
      4  : PunkteEingabe (Anfangszeiger);
      5  : Sortieren_durch_Einfuegen (Anfangszeiger);
      6  : StartListeZeigen (Anfangszeiger);
      else begin
             Gotoxy (35,20); write ('Unzulaessige Eingabe');
             delay (500);
           end;
    end;
    Auswahlmaske;
    Gotoxy (33,20); ClrEol; readln (Auswahl);
  until Auswahl = 0;
end.
```

Hinweis:
Damit Sie die Programmentwicklung am Rechner nachvollziehen können, geben Sie die Variablenvereinbarung, die Prozedur Auswahlmaske und das Hauptprogramm ein.
Entsprechend dem Lernfortschritt fügen Sie die behandelten Prozeduren in das "Rahmenprogramm" ein. Auf diese Weise komplettieren Sie das Programm.

9.2.2. Aufbau einer verketteten Liste durch Anfügen

Das Anfügen der Datensätze auf dem Heap und ihre Verkettung erfolgt stets in der Weise, wie es im Kapitel 9.1 dargestellt wurde. Man kann den Algorithmus somit auch als Standard-Anfüge-Algorithmus bezeichnen. Bei unterschiedlichen Aufgaben treten lediglich unterschiedliche Variablenbezeichnungen auf.

Prozedur Eingabe

Die Eingabe der Teilnehmerdaten wird in einer Prozedur mit dem Namen Eingabe behandelt, dessen Struktur nachstehend dargestellt ist. Zum Eintrag auf den Heap werden die Sportlerdaten an die Prozedur Anfuegen übergeben. Die Eingabe wird so lange wiederholt, bis die Abfrage auf weitere Eingaben mit n(ein) beantwortet wird. Der Algorithmus zum Hauptprogramm hat somit folgenden Aufbau:

Algorithmus

```
Algorithmus Eingabe (var Kopf : Zeiger)
    Eingabe Sportlerdaten
        Anfuegen (Kopf, Sportlerdaten )
    Eingabe Weiter
bis Weiter = 'n'
Ende
```

Programm

```
 1  procedure Eingabe (var E_Kopf : Zeiger);
 2  var Sportler : Teilnehmersatz;
 3  begin
 4    repeat
 5      ClrScr; Gotoxy (1,10);
 6      with Sportler do
 7      begin
 8        write ('Eingabe Start-Nummer '); readln (Nr);
 9        write ('Eingabe Name: '); readln (Name);
10        write ('Eingabe erreichte Leistung :'); readln (Punkte);
11      end;
12      writeln;
13      Anfuegen (E_Kopf, Sportler);
14      write ('Eingabe weiter (j/n) '); readln (Weiter);
15    until Weiter in ['n','N'];
16  end;
```

Erläuterungen

Die Prozedur wird mit

Eingabe (Anfangszeiger);

aus dem Hauptprogramm aufgerufen. Obwohl sich in der Prozedur keine Anweisung unmittelbar auf die Zeigervariable bezieht, muß Anfangszeiger als **Variablenparameter** an den formalen Parameter E_Kopf übergeben werden. Einerseits wird die Zeigervariable für die Prozedur Anfuegen benötigt. Andererseits muß die Anfangsadresse der verketteten Liste auch für die weiteren Funktionen zur Verfügung stehen. Deshalb muß auch im Aufruf der Prozedur Anfuegen

13 Anfuegen (E_Kopf, Sportler);

E_Kopf als Variablenparameter an die Prozedur Anfuegen übergeben werden.

Prozedur Anfügen

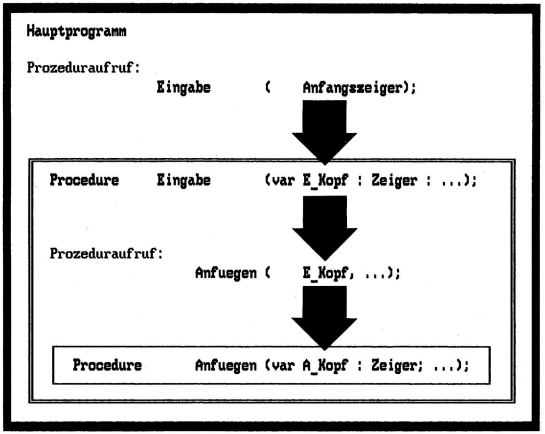
Der Algorithmus zum Eintrag des Datensatzes auf dem Heap und seine Verkettung wurde auf Seite 266 beschrieben. Im Unterschied zum obigen Algorithmus trägt die Zeigervariable, in der die Anfangsadresse der verketteten Liste vermerkt ist, den Namen A_Kopf.

Programm

```
1  procedure Anfuegen (var A_Kopf  : Zeiger;
                       Tnehmer : Teilnehmersatz);
2  var Laufzeiger, Hilfszeiger : Zeiger;
3  begin
4    new (Hilfszeiger);
5    Hilfszeiger^.Naechster := NIL;
6    Hilfszeiger^.Starter:= Tnehmer;
7    if A_Kopf = NIL
8      then begin
9        A_Kopf := Hilfszeiger;
10     end
11     else begin
12       Laufzeiger := A_Kopf;
13       while Laufzeiger^.Naechster <> NIL do
14         Laufzeiger := Laufzeiger^.Naechster;
15       Laufzeiger^.Naechster := Hilfszeiger;
16     end;
17 end;
```

Erläuterungen

Die für das Anfügen benötigten Daten erhält die Prozedur Anfügen aus der Prozedur Eingabe. Da die Zeigervariable E_Kopf als Variablenparameter auf A_Kopf übertragen wurde, beziehen sich die Wertzuweisungen auf A_Kopf letztlich auf die globale Variable Anfangszeiger. Das veranschaulicht auch die gegenüberliegende Abbildung:



Durch dieses "Weiterreichen" von Anfangszeiger als Variablenparameter über E_Kopf an A_Kopf wird erreicht, daß die Anfangsadresse der verketteten Liste in Anfangszeiger gebildet wird.

9.2.3 Ausgabe einer verketteten Liste

Die Ausgabe der Datensätze, die in einer verketteten Liste auf dem Heap gespeichert sind, erfolgt in der Prozedur StartListeZeigen. Dazu ist der Prozedur in einer Zeigervariablen die Adresse des ersten Datensatzes auf dem Heap mitzuteilen. Wird die Zeigervariable Lauf genannt, so werden mittels

writeln (Lauf^. Datenelement_1, ..., Lauf^. Datenelement_n);

die Elemente, aus denen der Datenteil besteht, ausgegeben.
Die Adresse des folgenden Datensatzes ist im Zeigerteil des Datensatzes enthalten, auf den Lauf gerade zeigt. Durch die Wertzuweisung

Lauf: = Zeigerteil der Bezugsvariablen von Lauf

wird die Adresse des nachfolgenden Satzes in der Zeigervariablen Lauf bereitgestellt. Die Ausgabe des Datensatzes wird dann mit der obigen WRITELN-Anweisung vorgenommen.
Diese beiden Anweisungen müssen so lange durchgeführt werden, bis der letzte Datensatz ausgegeben worden ist. Der letzte Datensatz ist daran zu erkennen, daß in dessen Zeigerteil der Wert NIL enthalten ist. In der Form eines Struktogramms lautet der Algorithmus:

Algorithmus

Prozedur StartListeZeigen (Lauf : Zeiger)
Solange Lauf ungleich NIL
Ausgabe des Datenteils der Bezugsvariablen von Lauf
Lauf := Zeigerteil der Bezugsvariablen von Lauf
Ende

Programm

```
 1  procedure StartListeZeigen (Lauf : Zeiger);
 2  begin
 3    while Lauf <> NIL do
 4    begin
        with Lauf^.Starter do
 5        writeln (Nr:4,' ',Name,' ':27-length(Name), Punkte:10);
 6      Lauf := Lauf^.Naechster;
 7    end;
 8    writeln; write ('Weiter <RETURN> druecken ');
 9    read (Weiter);
10  end;
```

Die Prozedur wird aus dem Hauptprogramm aufgerufen durch

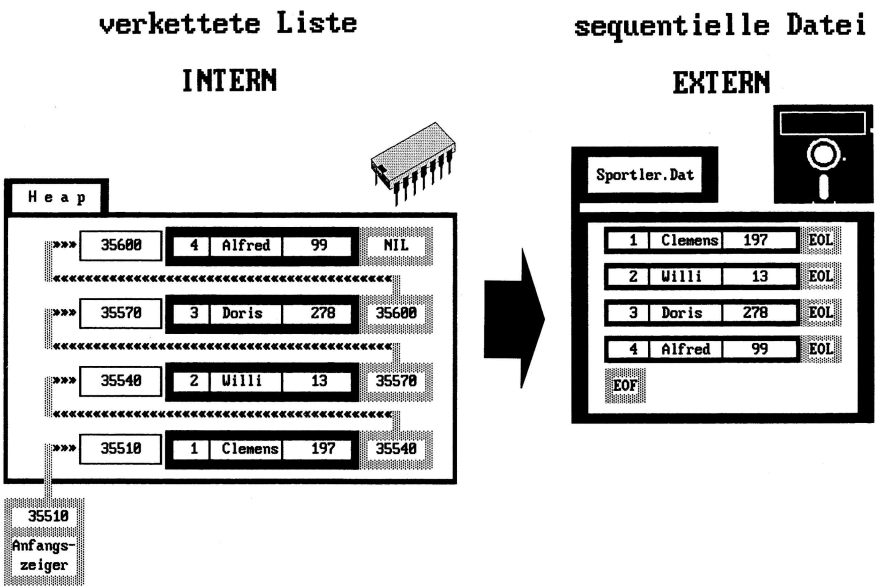
StartListeZeigen (Anfangszeiger);

Erläuterungen
Beim Aufruf der Prozedur muß ihr die Anfangsadresse der verketteten Liste mitgeteilt werden. Dazu wird die Zeigervariable Anfangszeiger als Werteparameter an den formalen Parameter Lauf übergeben.

Ergänzen Sie das obige "Rahmen-Programm" um die Prozeduren Eingabe, Anfuegen und StartListeZeigen und testen Sie es! Bitte beachten Sie, daß die Prozedur Anfuegen vor der Prozedur Eingabe stehen muß, da Anfuegen aus Eingabe aufgerufen wird.

9.2.4 Speichern einer verketteten Liste in eine sequentielle Datei

Zur Datensicherung sind die im Heap gespeicherten Datensätze in eine Datei auf einem externen Speicher zu schreiben. Dabei ist der Speicherplatz des Datensatzes, der in der Datei gespeichert worden ist, auf dem Heap wieder freizugeben, damit es bei einem späteren Einlesen aus der Datei nicht zu "Komplikationen" kommt.


verkettete Liste
INTERN
sequentielle Datei
EXTERN
Heap
35600
4
Alfred
99
NIL
35570
3
Doris
278
35600
35540
2
Willi
13
35570
35510
1
Clemens
197
35540
35510
Anfangs-
zeiger
Sportler.Dat
1
Clemens
197
EOL
2
Willi
13
EOL
3
Doris
278
EOL
4
Alfred
99
EOL
EOF

Für das sequentielle Speichern und das anschließende Löschen der Datensätze werden zwei Zeigervariablen benötigt. In einer Zeigervariablen, LaufZeiger genannt, steht die Adresse des zu speichernden Satzes. In einer weiteren Zeigervariablen mit dem Namen LoeschZeiger wird die Adresse des Datensatzes geführt, der gespeichert worden ist und dessen Platz nun freigegeben werden kann.
Ablauf der Prozedur: Zunächst wird die Datei auf dem externen Speicher eingerichtet und zum Schreiben geöffnet. Anschließend wird der erste Datensatz auf dem Heap in die Datei eingetragen.
Damit der Speicherplatz des zu löschenden Datensatzes bekannt ist, wird durch die Wertzuweisung

```
LoeschZeiger: = LaufZeiger
```

in der Zeigervariablen LoeschZeiger die Heap-Adresse des gespeicherten Datensatzes vermerkt. Bevor der Datensatz endgültig gelöscht wird, muß der Zeigerteil, in dem die Adresse des Folgesatzes steht, der Zeigervariablen LaufZeiger zugewiesen werden:

```
LaufZeiger: = LaufZeiger^. Naechster;
```

Anschließend wird durch Dispose (LoeschZeiger) der Speicherplatz des Datensatzes freigegeben.
Damit ist die Speicher- und Löschoperation für den ersten Datensatz durchgeführt. Zur Behandlung des nächsten Datensatzes wird entsprechend verfahren.
Dieser Vorgang wird so lange wiederholt, bis der letzte Datensatz, in dessen Zeigerteil der Wert NIL steht, in der Datei gespeichert und auf dem Heap gelöscht ist.

Algorithmus

Prozedur ListeInDateiSchreiben (LaufZeiger: Zeiger)
assign (SportDat,'Sportler.Dat')
rewrite(SportDat)
Solange LaufZeiger ungleich NIL
Datenteil der Bezugsvariablen von LaufZeiger in Datei SportDat schreiben
LoeschZeiger := LaufZeiger
Laufzeiger := Zeigerteil der Bezugsvariablen von LaufZeiger
DISPOSE (LoeschZeiger)
Schliepe Datei SportDat
Ende

Programm

```
procedure ListeInDateiSchreiben (LaufZeiger : Zeiger);
var LoeschZeiger : Zeiger;
    SportDat     : File of Teilnehmersatz;
begin
  assign(SportDat,'Sportler.Dat');
  rewrite(SportDat);
  while LaufZeiger <> Nil do
  begin
    write(SportDat,LaufZeiger^.Starter);
    LoeschZeiger := LaufZeiger;
    LaufZeiger := LaufZeiger^.Naechster;
    dispose(LoeschZeiger);
  end;
  close(SportDat);
end;
```

Die Prozedur wird aus dem Hauptprogramm aufgerufen mittels

ListeInDateiSchreiben (Anfangszeiger);

9.2.5 Lesen einer Datei mit Aufbau einer verketteten Liste

Die in der Datei gespeicherten Daten werden nun in den Arbeitsspeicher gelesen. Dabei ist auf dem Heap die verkettete Liste wieder aufzubauen. Zum Lesen der Datensätze wird die Prozedur Dateilesen_Listenaufbau entwickelt.
Zum Aufbau der verketteten Liste greifen wir auf die Prozedur Anfuegen zurück. An sie müssen aus der Prozedur Dateilesen_Listenaufbau die Zeigervariable, die die Anfangsadresse auf dem Heap enthält, und der Datensatz übergeben werden.
Damit die Zeigervariable Anfangszeiger aus der Prozedur Dateilesen_Listenaufbau an die Prozedur Anfuegen übergeben werden kann, wird Anfangszeiger aus dem Hauptprogramm als Variablenparameter an die Prozedur Dateilesen_Listenaufbau übergeben und von dort an die Prozedur Anfuegen "weitergereicht".
Die Datei wird sequentiell gelesen. Deshalb kann der Algorithmus, der im Kapitel 5.2.1 besprochen wurde, eingesetzt werden.

Algorithmus

```
Algorithmus Dateilesen_Listenaufbau          (var Kopf : Zeiger)
Assign (SportDat, 'Sportler.Dat')
Reset (SportDat )
Solange nicht EOF (SportDat)
    Lesen Datensatz
    Anfuegen (Kopf, Datensatz )
Close (SportDat)
Ende
```

Programm

```
procedure Dateilesen_Listenaufbau(var D_Kopf : Zeiger);
var  SportDat : file of Teilnehmersatz;
     TnSatz   : Teilnehmersatz;
begin
  assign(SportDat,'Sportler.Dat');
  reset(SportDat);
  while not EOF(SportDat) do
  begin
    read(SportDat,TnSatz);
    Anfuegen (D_Kopf, TnSatz);
  end;
  close(SportDat);
end;
```

Erläuterungen

Die Prozedur wird im Hauptprogramm unter Auswahlpunkt 3 aufgerufen.

```
3: begin
   Anfangszeiger: = NIL;
   Dateilesen_Listenaufbau (Anfangszeiger);
   end;
```

Vor ihrem Aufruf ist der Anfangszeiger mit dem Wert NIL zu belegen, da sonst der Aufbau der Kette nicht korrekt verlaufen würde. An dem Wert von Anfangszeiger wird nämlich in der Prozedur Anfuegen festgestellt, ob ein Dateneintrag der erste Datensatz auf dem Heap ist.
Die Zeigervariable Anfangszeiger wird als Variablenparameter an die Prozedur Dateilesen_Listenaufbau übergeben. Dadurch wird sichergestellt, daß nach der Bearbeitung der Prozedur die Anfangsadresse der verketteten Liste in Anfangszeiger zur Verfügung steht.

9.2.6 Suchen eines Datensatzes in einer verketteten Liste

Nach Beendigung eines Wettkampfteils sind die Punkte, die die Teilnehmer erreicht haben, zu den schon vorhandenen zu addieren.

Der Arbeitsablauf ist so zu gestalten, daß der Bediener die Starternummer und die Punktzahl eingibt. Die Eingabe ist beendet, wenn für Starternummer eine Null eingegeben wurde.
Das algorithmische Kernproblem dieser Aufgabe besteht darin, den zur Startnummer gehörenden Datensatz in der verketteten Liste zu suchen. Dazu muß ab dem ersten Datensatz jeder Datensatz daraufhin untersucht werden, ob die Starternummer im Datensatz mit der eingegebenen Nummer übereinstimmt. Solange die Nummern ungleich sind, muß der jeweils folgende Datensatz angesprochen werden. Die Suchschleife ist also zu verlassen, wenn der Datensatz mit der entsprechenden Starternummer gefunden ist oder wenn das Endemerkmal NIL angetroffen wurde.
Nach Verlassen der Schleife ist zu prüfen, aus welchem Grund die Schleife beendet wurde. Stimmen die beiden Nummern überein, so ist der gewünschte Datensatz gefunden worden, und die eingegebenen Punkte können addiert werden. Liegt keine Übereinstimmung vor, so wurde die Schleife verlassen, weil das Endemerkmal NIL erreicht wurde. Somit gibt es in der verketteten Liste keinen Datensatz, der die eingegebene Starternummer aufweist.
Beim Aufbau des Algorithmus ist schließlich zu berücksichtigen, daß die Suchroutine nur durchlaufen werden darf, solange für Starternummer eine Zahl ungleich Null eingegeben wurde. Die Eingabe einer Null bedeutete ja nach der Aufgabenstellung, daß die Punkteeingabe beendet werden soll.

Algorithmus

Prozedur PunkteEingabe (P_Kopf : Zeiger)

Eingabe Startnummer

Solange Startnummer <> 0

HilfZeiger := P_Kopf

Solange (HilfZeiger <> NIL) und Startnummer im Datenteil der Bezugsvariablen von HilfZeiger <> eingegebener Startnummer

HilfZeiger := Zeigerteil der Bezugsvariablen von HilfZeiger

Startnummer im Datensatz = eingegebener Startnummer ?

ja	nein
Eingabe Punkte	
eingegebene Punkte hinzu-addieren	Fehlermeldung: "Startnummer nicht vorhanden"
Eingabe Startnummer	

Ende

Programm

```
procedure PunkteEingabe(P_Kopf : Zeiger);
var HilfZeiger          : Zeiger;
    Startnummer, Punkte : integer;
begin
  ClrScr;
  Gotoxy (5,10);
  write('Startnummer (0=ENDE): '); readln(Startnummer);
  while Startnummer <> 0 do
  begin
    HilfZeiger := P_Kopf;
    while (HilfZeiger <> Nil) and
          (HilfZeiger^.Starter.Nr <> Startnummer) do
      HilfZeiger := HilfZeiger^.Naechster;
    if Startnummer = Hilfzeiger^.Starter.Nr
      then begin
        ClrEol;
        write('Dazugewonnene Punkte : '); readln(Punkte);
        HilfZeiger^.Starter.Punkte := HilfZeiger^.Starter.Punkte
                                      + Punkte;
      end
      else writeln ('Startnummer nicht vorhanden');
    Gotoxy (5,10); ClrEol;
    write('Startnummer (0=ENDE): '); readln(Startnummer);
  end;
end;
```

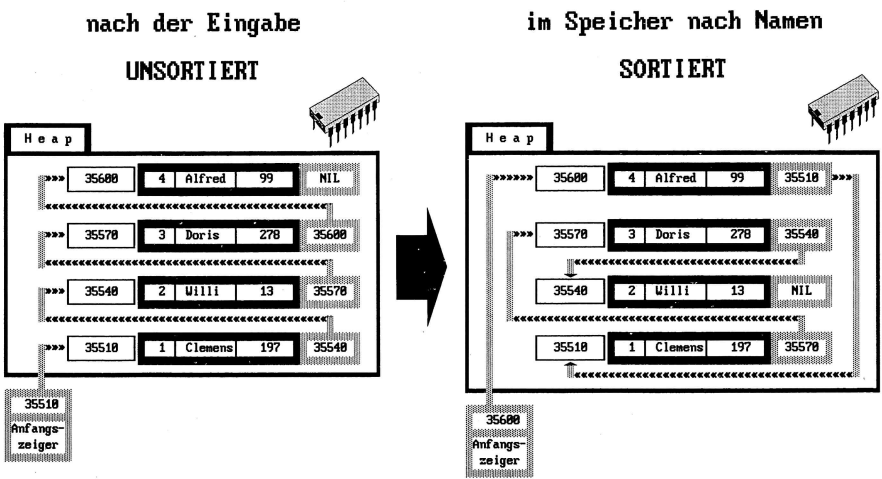
Erläuterungen
Die Prozedur benötigt für die Suchroutine die Anfangsadresse der verketteten Liste. Diese Information erhält sie durch den Prozeduraufruf aus dem Hauptprogramm:

```
PunkteEingabe (Anfangszeiger);
```

Dabei wird die Zeigervariable Anfangszeiger als Werteparameter an die Prozedur PunkteEingabe übergeben.
Ergänzen Sie das Programm! Testen Sie es, nachdem Sie durch Aufruf der Funktion 3 die Daten aus der Datei eingelesen haben! Speichern Sie anschließend die geänderten Daten mittels der Funktion 2 wieder ab!

9.2.7 Sortieren einer verketteten Liste

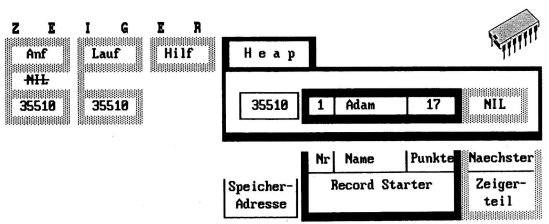
Nach Eingabe der Punkte ist die Siegerliste zu erstellen. Die Teilnehmerdaten sind dazu nach den erreichten Punkten in absteigender Reihenfolge zu sortieren.
Bei diesem Sortierlauf wird davon ausgegangen, daß die Teilnehmerdaten aus einer Datei gelesen werden. Beim Aufbau der verketteten Liste werden die Zeiger in der Weise gesetzt, daß die einzelnen Elemente in der logisch richtigen Reihenfolge verkettet werden.


nach der Eingabe
UNSORTIERT
Heap
35600
4
Alfred
99
NIL
35570
3
Doris
278
35600
35540
2
Willi
13
35570
35510
1
Clemens
197
35540
35510
Anfangs-
zeiger
im Speicher nach Namen
SORTIERT
Heap
35600
4
Alfred
99
35510
35570
3
Doris
278
35540
35540
2
Willi
13
NIL
35510
1
Clemens
197
35570
35600
Anfangs-
zeiger

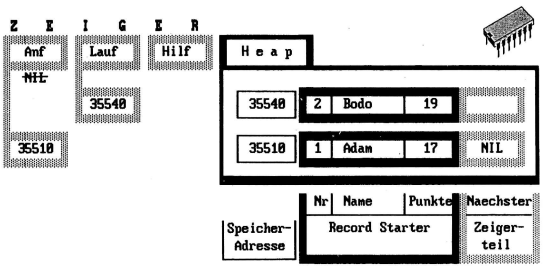
Die logisch richtige Reihenfolge wird dadurch erreicht, daß der Anfangszeiger auf den Datensatz mit der größten Punktzahl zeigt. Im Zeigerteil dieses Datensatzes ist die Adresse des Datensatzes mit der zweitgrößten Punktzahl vermerkt usw. Der Zeigerteil des Datensatzes mit der niedrigsten Punktzahl schließlich weist den Wert NIL auf.
Um uns die Logik des Sortieralgorithmus zu verdeutlichen, wird ein Beispielfall "durchgespielt". In der Datei seien folgende Datensätze gespeichert:

Nr	Name	Punkte
1	Adam	17
2	Bodo	19
3	Cosi	18
4	Dora	10
	EOF	

Vor dem Einlesen des ersten Datensatzes aus der Datei wird der Anfangszeiger, hier Anf genannt, zunächst auf NIL gesetzt. Mit new (Lauf) wird auf dem Heap für den Datensatz Speicherplatz reserviert, in den der erste Datensatz aus der Datei eingelesen wird. Dessen Zeigerteil erhält den Wert, der in der Zeigervariablen Anf steht. Anf seinerseits wird der Inhalt von Lauf zugewiesen. Damit beinhaltet Anf die "Einstiegsadresse" zur verketteten Liste.



Mit new (Lauf) wird ein weiterer Speicherbereich auf dem Heap reserviert, in den der zweite Datensatz eingetragen wird.



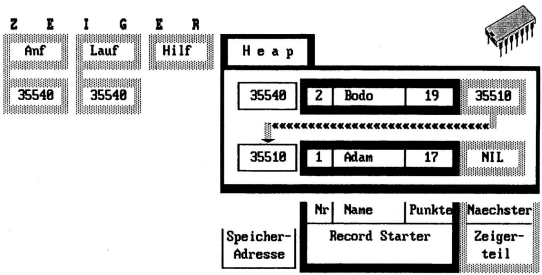
Die Punktzahl des Bodo-Datensatzes ist größer als die des Adam-Satzes. Zur richtigen Sortierung sind nun zwei Zeigeroperationen erforderlich. Zum einen muß der Zeiger des Satz mit der höchsten Punktzahl so gestellt werden, daß er auf den Satz mit der zweitgrößten Punktzahl zeigt. Das geschieht durch die Wertzuweisung

```
Lauf^. Naechster: = Anf;
```

Zum anderen muß Anf die Adresse des Datensatzes mit der größten Punktzahl erhalten. Das geschieht durch die Wertzuweisung

```
Anf: = Lauf;
```

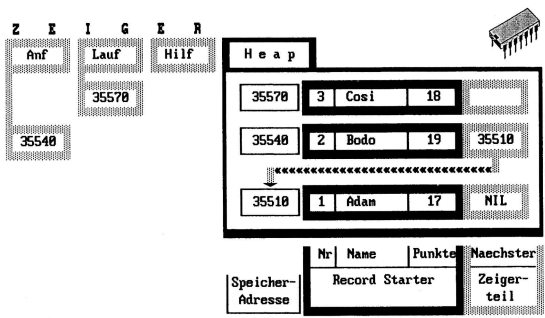
Mit diesen beiden Zeigeroperationen wurde der neue Datensatz **vorn eingefügt**. Der Wertbelegungsplan sieht wie folgt aus:



Um einen Datensatz logisch vorn anzufügen, sind somit folgende Programmschritte erforderlich:

```
if Anf^.Starter.Punkte < Lauf^.Starter.Punkte
 then begin
        Lauf^.Naechster := Anf;
        Anf := Lauf;
      end
```

Zur Aufnahme eines weiteren Datensatzes wird wiederum mit new (Lauf) Speicherbereich auf dem Heap reserviert, in den der nächste Datensatz (Cosi) eingelesen wird.



Die Punktzahl des Cosi-Satzes ist kleiner als die bislang höchste Punktzahl. Um zu prüfen, wie dieser Satz einzuordnen ist, wird die verkettete Liste durchlaufen. Ähnlich wie beim Suchen in einer verketteten Liste geschieht das in einer WHILE-Schleife.
Die WHILE-Schleife darf erst verlassen werden, wenn entweder der letzte Datensatz mit dem Zeigermerkmal NIL erreicht oder der aktuelle Datensatz eingefügt ist. Deshalb wird vor Eintritt in die Schleife eine boolsche Variable Eingefuegt auf false gesetzt. Ferner wird die "Einstiegsadresse", die in Anf gespeichert ist, in die Zeigervariable Hilf kopiert.
Innerhalb der WHILE-Schleife wird nun verglichen, ob die Punktzahl des Datensatzes mit der nächst kleineren Punktzahl (Adam-Satz) kleiner ist als die des zuletzt eingetragenen Cosi-Satzes. Der Zugriff auf die Punktzahl des Adam-Satzes erfolgt über den Zeigerteil des Bodo-Satzes. Da die Zeigervariable Hilf auf den Bodo-Satz zeigt, wird mittels

```
Hilf^. Naechster^. Starter. Punkte
```

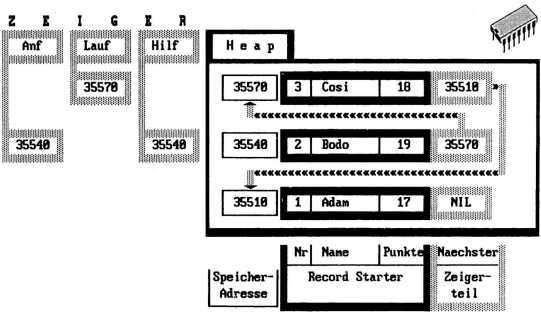
auf die Punktzahl des Adam-Satzes zugegriffen.
Der Vergleich der Punktzahl des Cosi-Satzes mit der Punktzahl des Adam-Satzes wird in der IF-Anweisung

```
if Hilf ^. Naechster ^. Starter. Punkte < Lauf ^. Starter. Punkte
```

durchgeführt. Da Adam weniger Punkte als Cosi besitzt, muß Cosi logisch vor Adam eingefügt werden. Dazu muß der Cosi-Zeigerteil den Wert des Bodo-Zeigerteils erhalten. Im Bodo-Zeigerteil muß die Adresse des Cosi-Satzes, die noch in Lauf steht, vermerkt werden.
Auf das Zeigerfeld des Bodo-Satzes wird durch Hilf^. Naechster zugegriffen. Der Zeigerteil des Cosi-Satzes wird durch Lauf^. Naechster angesprochen. Somit wird durch die Zeigeroperationen

```
Lauf^. Naechster: = Hilf^. Naechster;
Hilf^. Naechster: = Lauf
```

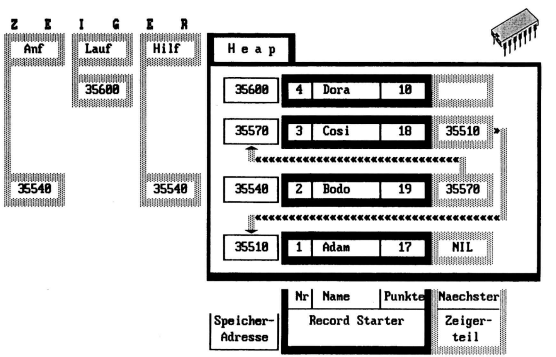
die logisch richtige Reihenfolge hergestellt. Das zeigt auch die nebenstehende Wertbelegungstabelle:



Dieses **Einfügen im Innern** der verketteten Liste findet, wie oben bereits dargestellt, in einer WHILE-Schleife statt. Damit die WHILE-Schleife verlassen werden kann, muß nach der Umsetzung der Zeiger die logische Variable Eingefuegt auf true gesetzt werden.
Die obigen Programmschritte, in denen das Einfügen am Anfang behandelt wurde, können wir nun um die Anweisungen zum Einfügen im Innern erweitern.

```
1  if Anf^.Starter.Punkte < Lauf^.Starter.Punkte
2  then begin
3    Lauf^.naechster := Anf;
4    Anf := Lauf;
5  end
6  else begin                              (* Einfügen im Innern *)
7    Hilf := Anf;
8    Eingefuegt := false;
9    while (Hilf^.naechster <> NIL) and (not Eingefuegt) do
10     if Hilf^.naechster^.Starter.Punkte < Lauf^.Starter.Punkte
11       then begin
12         Lauf^.naechster := Hilf^.naechster;
13         Hilf^.naechster := Lauf;
14         Eingefuegt := true;
15       end
16       else Hilf := Hilf^.naechster;     (* Weiterschalten *)
```

In unserem Beispiel konnte der Cosi-Satz logisch an der zweiten Stelle eingefügt werden. Falls ein Datensatz erst weiter im Innern der Liste eingeordnet werden kann, muß die Zeigervariable Hilf auf den jeweils logisch nächsten Datensatz weitergeschaltet werden. Das geschieht durch die Anweisung in Zeile 16. Ihre Bedeutung wird bei der Einordnung des nächsten Datensatzes noch erläutert.
Weil das Datei-Ende noch nicht erreicht ist, wird auf dem Heap erneut durch new (Lauf) Speicherplatz reserviert, in den der nächste Satz (Dora) eingelesen wird.



Die Punktzahl des Datensatzes Dora ist kleiner als die bislang höchste Punktzahl. Daher muß zur richtigen Einordnung der Zeiger die Liste durchlaufen werden. Von den oben dargestellten Programmschritten werden die erste Anweisung sowie die Anweisungen 6 bis 11 angesprochen. Wird in der IF-Anweisung

10 if Hilf^. Naechster^. Starter. Punkte < Lauf^. Starter. Punkte

festgestellt, daß die Punktzahl des Cosi-Satzes nicht kleiner ist als die des Dora-Satzes, muß der Datensatz mit der nächst kleineren Punktzahl herangezogen werden. Durch

Hilf: = Hilf ^. Naechster

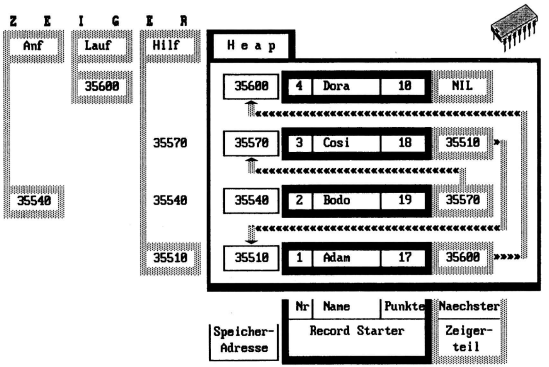
wird die Zeigervariable Hilf auf den Datensatz mit der nächst kleineren Punktzahl gesetzt und zeigt nun auf Cosi-Satz.
Da der Zeigerteil des Cosi-Satzes ungleich NIL ist und auch die boolsche Variable Eingefuegt auf false steht, wird die WHILE-Schleife erneut durchlaufen. In der IF-Anweisung in Zeile 10 werden nun die Punktzahlen des Adam-Satzes und des Dora-Satzes verglichen. Da der Dora-Satz weniger Punkte aufweist, ist die Zeigervariable Hilf auf den Datensatz mit der nächst kleineren Punktzahl zu setzen. Durch

```
Hilf: = Hilf^. Naechster
```

bildet sich in Hilf nun der Adreßwert 35510. Er zeigt somit auf den Adam-Satz.
Der Zeigerteil des Adam-Satzes weist den Wert NIL auf. Also ist das bislang logische Ende der Liste erreicht. Damit ist festgestellt, daß der zuletzt eingetragene Dora-Satz logisch an das Ende der Liste gesetzt werden muß. Das geschieht durch die Zeigeroperationen

```
Lauf^. Naechster: = Hilf^. Naechster;
Hilf^ Naechster: = Lauf
```

Das Ergebnis der Zeigeroperationen zeigt die nachstehende Wertbelegungstabelle:



Um einen Datensatz **am Ende einfügen** zu können, sind die obigen Programmschritte zu ergänzen.

```
1  if Anf^.Starter.Punkte < Lauf^.Starter.Punkte
2    then begin
3      Lauf^.naechster := Anf;
4      Anf := Lauf;
5    end
6    else begin                              (* Einfügen im Innern *)
7      Hilf := Anf;
8      Eingefuegt := false;
9      while (Hilf^.naechster <> NIL) and (not Eingefuegt) do
10       if Hilf^.naechster^.Starter.Punkte < Lauf^.Starter.Punkte
11         then begin
12           Lauf^.naechster := Hilf^.naechster;
13           Hilf^.naechster := Lauf;
14           Eingefuegt := true;
15         end
16         else Hilf := Hilf^.naechster;
17     if not Eingefuegt                     (* Einfügen am Ende *)
18       then begin
19         Lauf^.naechster := Hilf^.naechster;
20         Hilf^.naechster := Lauf;
21       end;
22   end;                                    (* of else in Zeile 6 *)
```

Damit haben wir den Kern des Algorithmus Sortieren durch Einfügen herausgearbeitet. Diese Anweisungen sind nun in den Zusammenhang mit den erforderlichen Datei-Operationen zu setzen. Das wird im nebenstehenden Struktogramm dargestellt:

Algorithmus

```
Prozedur Sortieren_durch_Einfuegen (var Sort_Kopf : Zeiger)
Datei zuordnen und Datei öffnen
Anf := NIL
wenn nicht Datei-Ende
  ja                                   | nein
  new (Lauf)                           | ./.
  Lese Datensatz in Lauf^.Starter      |
  Lauf^.naechster := Anf               |
  Anf := Lauf                          |
Solange nicht Datei-Ende
  new (Lauf)
  Lese Datensatz in Lauf^.Starter
  Wenn Punktzahl Anf^.Starter < Punktzahl Lauf^.Starter
    ja                                 | nein
    Lauf^.naechster:= Anf              | Hilf := Anf
    Anf := Lauf                        | Eingefuegt := false
                                       | Solange Hilf^.naechster <> NIL und
                                       | nicht Eingefuegt
                                       |   wenn Punktzahl Hilf^.naechster^.
                                       |   Starter < Punktzahl Lauf^.Starter
                                       |     ja                | nein
                                       |     Lauf^.naechster :=| Hilf :=
                                       |     Hilf^.naechster   | Hilf^.naechster
                                       |     Hilf^.naechster :=|
                                       |     Lauf              |
                                       |     Eingefuegt := true|
                                       |   wenn nicht Eingefuegt
                                       |     ja                | nein
                                       |     Lauf^.naechster := Hilf^.naechster | ./.
                                       |     Hilf^.naechster := Lauf            |
Sort_Kopf := Anf
Ende
```

Programm

```
procedure Sortieren_durch_Einfuegen (var Sort_Kopf : Zeiger);
var Anf,Hilf,Lauf : Zeiger;
    Eingefuegt    : boolean;
    SportDat      : file of Teilnehmersatz;
begin
  assign (SportDat,'Sportler.Dat');
  reset (SportDat);
  Anf:= NIL;
  if not eof (SportDat)
    then begin
      new (Lauf);
      read (SportDat, Lauf^.Starter);
      Lauf^.naechster := Anf;
      Anf := Lauf;
    end;
  while not eof (SportDat) do
  begin
    new (Lauf);
    read (SportDat, Lauf^.Starter);
    if Anf^.Starter.Punkte < Lauf^.Starter.Punkte
      then begin
        Lauf^.naechster := Anf;
        Anf := Lauf;
      end
      else begin
        Hilf := Anf;
        Eingefuegt := false;
        while (Hilf^.naechster <> NIL) and (not Eingefuegt) do
          if Hilf^.naechster^.Starter.Punkte > Lauf^.Starter.Punkte
            then begin
              Lauf^.naechster := Hilf^.naechster;
              Hilf^.naechster := Lauf;
              Eingefuegt := true;
            end
            else Hilf := Hilf^.naechster;
          if not Eingefuegt
            then begin
              Lauf^.naechster := Hilf^.naechster;
              Hilf^.naechster := Lauf;
            end;
      end;                                    (* of else *)
  end;                                        (* of while *)
  Sort_kopf := Anf;
end;
```

Die Prozedur wird aus dem Hauptprogramm durch

Sortieren_durch_Einfuegen (Anfangszeiger);

aufgerufen. Ergänzen Sie das Sportverwaltungsprogramm um die Prozedur und testen Sie es!
Natürlich kann die Prozedur Sortieren_durch_Einfügen auch bei der Ersteingabe von Daten eingesetzt werden. Das hat den Vorteil, daß die Datensätze unmittelbar nach Abschluß der Eingabe in sortierter Form vorliegen.

9.2.8 Zusammenstellung der Prozeduren zum Sportverwaltungsprogramm

Mit der Entwicklung der obigen Prozeduren haben wir die wesentlichen Programmbausteine geschaffen, die für das Sportverwaltungsprogramm benötigt werden. Zur besseren Übersicht werden die Prozedurköpfe und das Hauptprogramm zusammengestellt.

```
program Sportverwaltungsprogramm;
uses CRT;              (* 4 *)
type str25             = string (.25.);
     Teilnehmersatz = Record
                        Nr     : Integer;
                        Name   : str25;
                        Punkte : Integer;
                      end;
     Zeiger         = ^StartListe;
     StartListe     = record
                        Starter   : Teilnehmersatz;
                        Naechster : Zeiger;
                      end;

var  Anfangszeiger : Zeiger;
     Weiter        : char;
     Auswahl       : Integer;


procedure Anfuegen                    (var A_Kopf   : Zeiger;
                                           Tnehmer  : Teilnehmersatz);
 ...
procedure StartListeDrucken           (Lauf         : Zeiger);
 ...
Procedure ListeInDatei                (LaufZeiger   : Zeiger);
 ...
```

```
Procedure Dateilesen_Listenaufbau   (var D_Kopf    : Zeiger);
 ...
procedure PunkteEingabe             (P_Kopf       : Zeiger);
 ...
procedure Sortieren_durch_Einfuegen (var Sort_Kopf : Zeiger);
 ...
procedure Eingabe                   (var E_Kopf    : Zeiger);
 ...
procedure Auswahlmaske;
 ...

begin                                    (* HAUPTPROGRAMM*)
  Anfangszeiger:= NIL;
  Bildschirmmaske;
  Gotoxy (33,20); ClrEol; readln (Auswahl);
  repeat
    case Auswahl of
      1  : Eingabe (Anfangszeiger);
      2  : ListeInDatei(Anfangszeiger);
      3  : begin
             Anfangszeiger := NIL;
             Dateilesen_Listenaufbau(Anfangszeiger);
           end;
      4  : PunkteEingabe (Anfangszeiger);
      5  : Sortieren_durch_Einfuegen (Anfangszeiger);
      6  : StartListeZeigen(Anfangszeiger);
      else begin
             Gotoxy (35,20); write ('Unzulaessige Eingabe');
             delay (500);
           end;
    end;
    Auswahlmaske;
    Gotoxy (33,20); ClrEol; readln (Auswahl);
  until Auswahl = 0;
end.
```

Mit dem Datentyp Zeiger und seiner Anwendung auf verkettete Listen haben Sie eine Datenstruktur kennengelernt, die es in anderen Programmiersprachen, wie z.B. BASIC oder COBOL nicht gibt. Auf der Grundlage der "Verzeigerung" von Datensätzen lassen sich nun noch weitere interessante Datenstrukturen wie doppelt verkettete Liste oder auch Baumstrukturen konstruieren. Diese recht reizvollen, aber auch sehr komplexen Algorithmen können hier nicht behandelt werden. Der interessierte Leser sei auf die umfangreiche einschlägige Literatur verwiesen.

9.2.9 Zusammenfassung

- Beim Aufbau einer verketteten Liste kann ein Datensatz logisch am Anfang, am Ende oder im Innern der Liste eingefügt werden.
- Soll ein Datensatz jeweils am Ende der Liste angefügt werden, kann folgende Prozedur verwendet werden:

```
procedure Anfuegen (var Kopf  : Zeiger;
                        Daten : Datensatz);
var Lauf, Hilf : Zeiger;
begin
  new (Hilf);
  Hilf^.Zeigerteil:= NIL;
  Hilf^.Datenteil := Daten;
  if Kopf = NIL
    then Kopf := Hilf
    else begin
      Lauf := Hilf;
      while Lauf^.Zeigerteil <> NIL do
        Lauf := Lauf^.Zeigerteil;
      Lauf^.Zeigerteil := Hilf;
    end;
end;
```

Im Aufruf der Prozedur sind der Anfangszeiger sowie die Daten als Parameter anzugeben.

- Sollen Datensätze logisch am Anfang der Liste eingefügt werden, ist folgende Prozedur zu verwenden:

```
procedure Vorn_Einfuegen (var Kopf  : Zeiger;
                              Daten : Datensatz);
var Hilf : Zeiger;
begin
  New (Hilf);
  Hilf^.Datenteil := Daten;
  Hilf^.Zeigerteil := NIL;
  if Kopf = NIL
    then Kopf := Hilf
    else begin
      Hilf^.Zeigerteil := Kopf;
      Kopf := Hilf;
    end;
end;
```

Im Aufruf der Prozedur sind der Anfangszeiger sowie die Daten als Parameter anzugeben.

Ist ein Datensatz logisch an einer bestimmten Stelle in die verkettete Liste einzufügen, so eignet sich folgende Prozedur. Voraussetzung: Im Prozeduraufruf wird neben dem Anfangszeiger und den Daten auch die Nummer des Datensatzes angegeben, hinter den der neue Datensatz eingefügt werden soll.

```
procedure An_bestimmter_Stelle_Einfuege  (var Kopf   : Zeiger;
                                           Daten  : Datensatz;
                                           Stelle :integer);
var Hilf, Lauf : Zeiger;
    I          : integer;
begin
  new (Hilf);
  Hilf^.Zeigerteil := NIL;
  Hilf^.Datenteil := Daten;
  if Kopf = NIL
    then Kopf := Hilf
    else begin
      Lauf := Kopf
      for I := 1 to Stelle do
        if Lauf^.Zeigerteil <> NIL
          then Lauf := Lauf^.Zeigerteil;
      Hilf^.Zeigerteil := Lauf^.Zeigerteil;
      Lauf^.Zeigerteil := Hilf;
    end;
end;
```

Zur Ausgabe einer verketteten Liste kann folgende Prozedur eingesetzt werden:

```
procedure Ausgabe_Liste (Lauf  : Zeiger;
                         Daten : Datensatz);
begin
  while Lauf <> NIL do
  begin
    Ausgabeanweisung;
    Lauf := Lauf.Zeigerteil;
  end;
end;
```

Das Suchen nach einem Datensatz mit einem bestimmten Merkmal kann über die folgende Prozedur erfolgen:

```
procedure Suchen (    Kopf        : Zeiger;
                  var Daten       : Datensatz;
                      Suchmerkmal : Datentyp);
var Lauf : Zeiger;
begin
  Hilf := Kopf;
  while (Hilf <> NIL) and (Hilf^.Datenteil.Element <> Suchmerkmal do
    Lauf := Lauf^.Zeigerteil;
  if Hilf^.Datenteil.Element = Suchmerkmal
    then  (* Gefunden *)
    else  (* Datensatz mit Suchmerkmal nicht vorhanden *)
end;
```

Im Prozeduraufruf muß neben dem Anfangszeiger und den Daten auch das Suchmerkmal als Parameter angegeben werden.

9.2.10 Übungen

1. Aufgabe
Nach dem ersten Programmeinsatz wünscht der Veranstalter, daß die Punkteingabe verbessert wird. Nach seiner Ansicht sollten Starternummer und Teilnehmername der Reihe nach am Bildschirm angezeigt werden. Der Bediener braucht dann nur noch die erreichte Punktzahl einzugeben.
Überarbeiten Sie die Prozedur PunkteEingabe nach den Vorstellungen des Veranstalters!

2.Aufgabe
Da es vorkommt, daß Wettkampfteilnehmer vorzeitig den Wettkampf abbrechen, sind sie aus der Liste der Starter zu entfernen.
Entwickeln Sie eine Löschprozedur, mit der nach der Eingabe der Starternummer der Datensatz aus der Liste gelöscht wird. Ändern Sie zum Aufruf der Löschprozedur das Auswahlmenü und ergänzen Sie die CASE-Struktur um den Aufruf der Löschprozedur!

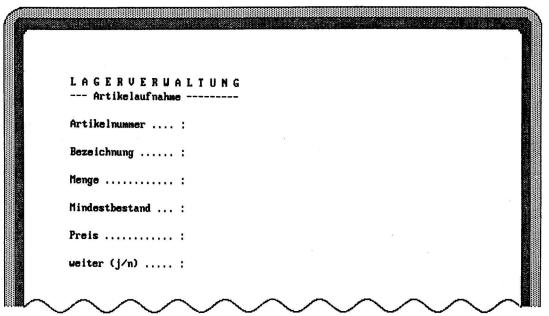
3. Aufgabe

Es ist ein Lagerverwaltungsprogramm unter Anwendung der Datenstruktur der verketteten Liste zu entwickeln. Der Artikelsatz ist wie folgt ausgebaut:

Artikelnummer
Artikelbezeichnung
Menge
Mindestbestand
Preis

Folgende Funktionen soll das Programm erfüllen:

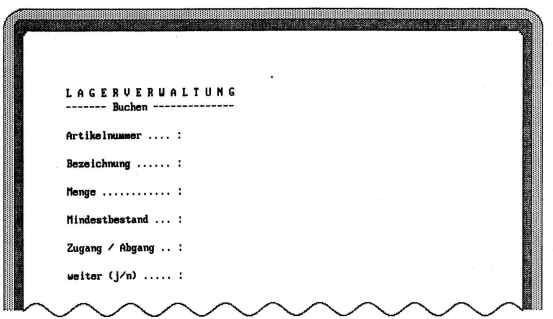
1. Ersteingabe der Artikeldaten:
   Bei der Ersteingabe sind die Artikeldaten sortiert nach Artikelnummer in die Liste einzutragen. Nach Abschluß der Eingabe sind die Artikeldaten in die Datei LagerL.Dat zu speichern.
   · Die Eingabe der Artikeldaten soll mittels folgender Bildschirmmaske erfolgen:



2. Hinzufügen weiterer Artikeldaten:
   Bei dieser Funktion wird davon ausgegangen, daß bereits Artikeldaten in der Datei LagerL.Dat vorhanden sind. Sie sind vor der Eingabe weiterer Daten auf den Heap einzulesen.

Für die Eingabe soll die obige Bildschirmmaske verwendet werden.
Nach Abschluß der Eingaben sind die Daten in die Datei LagerL.Dat zu speichern.

3. Buchen von Zu- und Abgängen:
Der Arbeitsablauf ist in der Weise zu gestalten, daß zunächst die Datei LagerL.Dat eingelesen wird. Anschließend ist die Artikelnummer einzugeben. Ist die Artikelnummer schon vorhanden, werden die weiteren Daten des Artikels angezeigt. Nach Eingabe der Mengenänderung wird der neue Bestand im Feld Menge ausgegeben.
Existiert die Artikelnummer nicht, soll ein Fehlerhinweis gegeben werden.
Beim Buchen ist folgende Bildschirmmaske zu verwenden:



Nach Abschluß sämtlicher Buchungen sind alle Daten in der Datei LagerL.Dat zu speichern.

4. Sortierung der Artikeldaten nach der Artikelbezeichnung:
Die Artikeldaten sind aus der Datei LagerL.Dat einzulesen und nach der Artikelbezeichnung sortiert auf dem Heap einzutragen. Anschließend sind die Artikeldaten auszugeben.

5. Sortierung der Artikeldaten nach der Artikelnummer:
Ablauf wie bei Punkt 4, Sortierung jedoch nach der Artikelnummer.

9.2.11 Lösungen

1. Aufgabe

```
procedure PunkteEingabe (Kopf : Zeiger);
var HilfZeiger : Zeiger;
    Punkte     : integer;
begin
  ClrScr;
  HilfZeiger := Kopf;
  while (HilfZeiger <> NIL) do
  begin
    Gotoxy (5,10); ClrEol;
    with HilfZeiger^.Starter do
      write (Nr:4, '  ', Name);
    Gotoxy (35,10); write ('erreichte Punkte:');
    readln (Punkte);
    HilfZeiger^.Starter.Punkte := HilfZeiger^.Starter.Punkte
                                  + Punkte;
    HilfZeiger := HilfZeiger^.Naechster;
  end;
end;
```

2. Aufgabe

```
procedure Datensatz_Loeschen (var Kopf : Zeiger);
var Lauf, Vorgaenger : Zeiger;
    StartNr          : integer;
begin
  ClrScr;
  Gotoxy (5,10); write ('Eingabe zu loeschende Startnummer: ');
  readln (StartNr);
  Lauf := Kopf;
  if Lauf^.Starter.Nr = StartNr
    then begin
      Kopf := Lauf^.Naechster;
      dispose (Lauf);
    end
    else begin
      while (Lauf <> NIL) and (Lauf^.Starter.Nr <> StartNr) do
      begin
        Vorgaenger := Lauf;
        Lauf := Lauf^.Naechster;
      end;
```

```
      if Lauf^.Starter.Nr = StartNr
        then begin
          Vorgaenger^.Naechster := Lauf^.Naechster;
          dispose (Lauf);
        end
        else begin
          writeln ('Startnummer nicht gefunden');
          write ('Weiter <RETURN> eingeben');
          read (Weiter);
        end;
    end;
end;


procedure Bildschirmmaske;
begin

  ...

  Gotoxy (5,17); writeln ('Teilnehmerdaten anzeigen .. 6');
  Gotoxy (5,19); writeln ('Teilnehmerdaten loeschen .. 7');
  Gotoxy (5,21); writeln ('Auswahl (0 = Ende) ..... ');
end;


begin  (* HAUPTPROGRAMM *)

  ...

    6  : StartListeDrucken(Anfangszeiger);
    7  : Datensatz_Loeschen (Anfangszeiger);

    ...

end.
```

3. Aufgabe

```
program Lagerverwaltung;
uses CRT;          {* 4 *}
type Artikelsatz = record
                     Nr                          :integer;
                     Bezeichnung                 : string (.30.);
                     Menge, Mindestbestand       : integer;
                     Preis                       : real;
                   end;
       Lagersatz = ^Artikel;
       Artikel   = record
                     Art       : Artikelsatz;
                     Naechster : Lagersatz;
                   end;

var Anfangzeiger : Lagersatz;
    ArtSatz      : Artikelsatz;
    Auswahl      : integer;
    LagDat       : file of Artikelsatz;


procedure A_Bildschirm;
begin
  ClrScr;
  Gotoxy (5,4);  write ('L A G E R V E R W A L T U N G');
  Gotoxy (5,5);  write ('--- Artikelaufnahme ---------');
  Gotoxy (5,7);  write ('Artikelnummer .... :');
  Gotoxy (5,9);  write ('Bezeichnung ..... :');
  Gotoxy (5,11); write ('Menge ........... :');
  Gotoxy (5,13); write ('Mindestbestand ... :');
  Gotoxy (5,15); write ('Preis ........... :');
  Gotoxy (5,17); write ('weiter (j/n) ..... :');
end;


procedure B_Bildschirm;
begin
  ClrScr;
  Gotoxy (5,4);  write ('L A G E R V E R W A L T U N G');
  Gotoxy (5,5);  write ('------- Buchen -------------');
  Gotoxy (5,7);  write ('Artikelnummer .... :');
  Gotoxy (5,9);  write ('Bezeichnung ..... :');
  Gotoxy (5,11); write ('Menge ........... :');
  Gotoxy (5,13); write ('Mindestbestand ... :');
  Gotoxy (5,15); write ('Zugang / Abgang .. :');
  Gotoxy (5,17); write ('weiter (j/n) ..... :');
end;


procedure Einfuegen_Sortiert_Nr (var Kopf  : Lagersatz;
                                     Daten : Artikelsatz);
```

```
var Anf, Hilf, Lauf : Lagersatz;
    Eingefuegt      : boolean;
begin
  Anf := Kopf;
  if Anf = NIL
    then begin          new (Lauf);
      Lauf^.Art := Daten;
      Lauf^.naechster := Anf;
      Anf := Lauf;
    end
    else begin
      new (Lauf);
      Lauf^.Art := Daten;
      if Anf^.Art.Nr > Lauf^.Art.Nr
        then begin
          Lauf^.naechster := Anf;
          Anf := Lauf;
        end
        else begin
          Hilf := Anf;
          Eingefuegt := false;
          while (Hilf^.naechster <> NIL) and (not Eingefuegt) do
            if Hilf^.naechster^.Art.Nr > Lauf^.Art.Nr
              then begin
                Lauf^.naechster := Hilf^.naechster;
                Hilf^.naechster := Lauf;
                Eingefuegt := true;
              end
              else Hilf := Hilf^.naechster;
          if not Eingefuegt
            then begin
              Lauf^.naechster := Hilf^.naechster;
              Hilf^.naechster := Lauf;
            end;
        end;                                  (* of else *)
    end;                                      (* of else *)
    Kopf := Anf;
end;


procedure Einfuegen_Sortiert_Name (var Kopf  : Lagersatz;
                                       Daten : Artikelsatz);
var Anf, Hilf, Lauf : Lagersatz;
    Eingefuegt      : boolean;
begin
  Anf := Kopf;
  if Anf = NIL
    then begin          new (Lauf);
      Lauf^.Art := Daten;
      Lauf^.naechster := Anf;
      Anf := Lauf;
    end
    else begin
```

```
      new (Lauf);
      Lauf^.Art := Daten;
      if Anf^.Art.Bezeichnung > Lauf^.Art.Bezeichnung
        then begin
          Lauf^.naechster := Anf;
          Anf := Lauf;
        end
        else begin
          Hilf := Anf;
          Eingefuegt := false;
          while (Hilf^.naechster <> NIL) and (not Eingefuegt) do
            if Hilf^.naechster^.Art.Bezeichnung >
               Lauf^.Art.Bezeichnung
              then begin
                Lauf^.naechster := Hilf^.naechster;
                Hilf^.naechster := Lauf;
                Eingefuegt := true;
              end
              else Hilf := Hilf^.naechster;
          if not Eingefuegt
            then begin
              Lauf^.naechster := Hilf^.naechster;
              Hilf^.naechster := Lauf;
            end;
        end;                                    (* of else *)
    end;                                        (* of else *)
    Kopf := Anf;
end;


procedure DateiLesen (var Kopf  : Lagersatz;
                      var Daten : Artikelsatz;
                          Wahl  : integer);
begin
  assign (LagDat, 'LagerL.Dat');
  reset (LagDat);
  while not eof (LagDat) do
  begin
    read (LagDat,Daten);
    if Wahl = 4
      then Einfuegen_Sortiert_Name (Kopf, Daten)
      else Einfuegen_Sortiert_Nr (Kopf, Daten);
  end;
  close (LagDat);
end;


procedure Artikel_Eingabe (var Kopf  : Lagersatz;
                           var Daten : Artikelsatz);
var Weiter : char;
begin
  A_Bildschirm;
  repeat
```

```
    with Daten do
    begin
      Gotoxy (26,7); ClrEol; readln (Nr);
      Gotoxy (26,9); ClrEol; readln (Bezeichnung);
      Gotoxy (26,11);ClrEol; readln (Menge);
      Gotoxy (26,13);ClrEol; readln (Mindestbestand);
      Gotoxy (26,15);ClrEol; readln (Preis);
      Einfuegen_Sortiert_Nr (Kopf, Daten);
    end;                                    (* of with *)
    Gotoxy (26,17);ClrEol; readln (Weiter);
  until Weiter in ('n','N');
end;


procedure ListeDrucken (Kopf : Lagersatz);
var Weiter : char;
begin
  ClrScr;
  writeln ('ArtNr  Bezeichnung              Menge   Mind.Best  Preis');
  writeln ('-------------------------------------------------------');
  while Kopf <> NIL do
  begin
    With Kopf^.Art do
      writeln (Nr:4,' ',Bezeichnung,' ':30-Length (Bezeichnung),
               Menge:5, Mindestbestand:8, Preis:13:2);
    Kopf := Kopf^.Naechster;
  end;
  writeln; write ('Weiter <RETURN> drcken ');
  read (Weiter);
end;


Procedure ListeInDatei (LaufZeiger : Lagersatz);
Var LoeschZeiger : Lagersatz;
Begin
  Assign(LagDat,'LagerL.Dat');
  ReWrite(LagDat);
  While LaufZeiger <> Nil do
  Begin
    Write(LagDat,Laufzeiger^.Art);
    LoeschZeiger := LaufZeiger;
    LaufZeiger := LaufZeiger^.Naechster;
    Dispose(LoeschZeiger);
  End;
  Close(LagDat);
End;


Procedure Buchen (Kopf : Lagersatz);
Var Hilf          : Lagersatz;
    ArtNr, EMenge : Integer;
    weiter        : char;
Begin
```

```
  B_Bildschirm;
  Repeat
    Gotoxy (26,7); ClrEol; read (ArtNr);
    Hilf := Kopf;
    while (Hilf <> Nil) and (Hilf^.Art.Nr <> Artnr) do
      Hilf := Hilf^.Naechster;
    If Hilf = Nil
      then begin
        Gotoxy (35,7);
        write('  Artikelnummer nicht vorhanden <RETURN>');
        readln (weiter);
      end
      else begin
        Gotoxy (26,9); ClrEol; write (Hilf^.Art.Bezeichnung);
        Gotoxy (26,11); write (Hilf^.Art.Menge:6);
        Gotoxy (26,13); write (Hilf^.Art.Mindestbestand:6);
        Gotoxy (26,15); ClrEol; readln (EMenge);
        Hilf^.Art.Menge := Hilf^.Art.Menge + EMenge;
        Gotoxy (26,11); write (Hilf^.Art.Menge:6);
      end;
    Gotoxy (26,17); ClrEol; readln (Weiter);
  Until Weiter in ['n','N'.];
End;


procedure MenuBildschirm;
begin
  ClrScr;
  Gotoxy (5,5);  writeln ('Lagerverwaltungsprogramm');
  Gotoxy (5,7);
  writeln ('Erst-Eingabe Artikeldaten ..................... 1');
  Gotoxy (5,9);
  writeln ('Hinzufügen Artikeldaten ....................... 2');
  Gotoxy (5,11);
  writeln ('Buchen ........................................ 3');
  Gotoxy (5,13);
  writeln ('Artikeldaten nach Bezeichnungen sortieren.. 4');
  Gotoxy (5,15);
  writeln ('Artikeldaten nach Art-Nr. sortieren ....... 5');
  Gotoxy (5,20);
  writeln ('Auswahl (0 = Ende) ........ ');
end;


begin                                    (* HAUPTPROGRAMM *)
  MenuBildschirm;
  Gotoxy (33,20); ClrEol; readln (Auswahl);
  repeat
    case Auswahl of
      1  : begin
             Anfangzeiger := Nil;
             Artikel_Eingabe (Anfangzeiger, ArtSatz);
             ListeDrucken (Anfangzeiger);
             ListeInDatei (Anfangzeiger);
```

```
            end;
        2  : begin
               Anfangzeiger := NIL;
               DateiLesen (Anfangzeiger, ArtSatz, Auswahl);
               Artikel_Eingabe (Anfangzeiger, ArtSatz);
               ListeDrucken (Anfangzeiger);
               ListeInDatei (Anfangzeiger);
             end;
        3  : begin
               Anfangzeiger := NIL;
               DateiLesen (Anfangzeiger, ArtSatz, Auswahl);
               Buchen (Anfangzeiger);
               ListeDrucken (Anfangzeiger);
               ListeInDatei (Anfangzeiger);
             end;
        4,5: begin
               Anfangzeiger := NIL;
               DateiLesen (Anfangzeiger, ArtSatz, Auswahl);
               ListeDrucken (Anfangzeiger);
             end;
        else begin
               Gotoxy (35,20); write ('Unzulaessige Eingabe');
               delay (500);
             end;
      end;                                        (* of case *)
      MenuBildschirm;
      Gotoxy (33,20); ClrEol; readln (Auswahl);
    until Auswahl = 0;
  end.
```

10 ANHANG

10.1 ASCII-Tabelle

ASCII-Code unter Turbo PASCAL (Industriestandard Set#2)

NULL	☺	☻	♦	♣	♠	BEL	BS	○	LF	♂	♀	CR	♫	✱	▶	◀	↕	‼
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼	BLANK	!	"	#	$	%	&	'
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	`	a	b	c
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
x	y	z	{	|	}	~	⌂	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï
120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139
î	ì	Ä	Å	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»	▒	▓	│	┤
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179
╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐	└	┴	┬	├	─	┼	╞	╟	╚
180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199
╔	╩	╦	╠	═	╬	╧	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌		
200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
				α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	∅	∈	∩
220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■					
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255				

10.2 Literaturverzeichnis

Borland International (Hg.): TURBO PASCAL DATABASE TOOLBOX, o. O. 1987

Gottfried, Byron S.: PROGRAMMIEREN IN PASCAL, Hamburg, New York 1986

Heimsoeth Software (Hg.): TURBO PASCAL 3.0 HANDBUCH, München 1985

derselbe: TURBO PASCAL 4.0, Band 1, BENUTZERHANDBUCH, München 1988
derselbe: TURBO PASCAL 4.0, Band 2, REFERENZHANDBUCH, München 1988
derselbe: TOOLBOX HANDBUCH, München 1984
Kaier, Ekkehard, und E. Rudolfs: TURBO PASCAL-WEGWEISER FÜR MICROCOMPUTER, Aufbaukurs, Braunschweig, Wiesbaden 1987
Kuhlmann, Gregor: INFORMATIK – FINANZMATHEMATIK, Bad Homburg vor der Höhe 1984
Kuhlmann, Gregor: QUICKMANUAL TURBO PASCAL, Rinteln 1987
Kuhlmann, Gregor: PROGRAMMIERSPRACHE TURBOPASCAL, Eine strukturierte Einführung, Reinbek 1988

Hinweis

Autor, Herausgeber und Verlag haben sämtliche Angaben, Hinweise und Beispiele, die in diesem Buch aufgeführt sind, sorgfältig geprüft. Dennoch können Fehler nicht völlig ausgeschlossen werden. Autor, Herausgeber und Verlag können deshalb keine Gewährleistung für die einwandfreie Funktion aller Angaben, Hinweise und Beispiele übernehmen. Für etwaige Folgeschäden an Geräten und Programmen, die durch die Benutzung der Inhalte dieses Buchs entstehen können, wird keine Haftung übernommen.
Für Hinweise auf Fehler und Verbesserungsvorschläge sind wir sehr dankbar.

Produkt- und Warennamen

CP/M ist ein eingetragenes Warenzeichen von Digital Research Inc.
IBM ist ein eingetragenes Warenzeichen von International Business Machines Corporation.
MS-DOS ist ein eingetragenes Warenzeichen von Microsoft Corporation.
Turbo-Pascal ist eingetragenes Warenzeichen von Borland Inc.
UNIX ist eingetragenes Warenzeichen von Bell Laboratories Inc.

10.3 Sachwortregister

.EXE 23
AddKey 183, 189f
AddRec 183, 188f
Adressierung, direkt 41
indiziert 41
Array 36ff, 59
Assign 116, 230

Bereichsüberprüfung 61
Bezugsvariable 247

Change dir 27
ClearKey 184, 193
Close 122, 230
CloseFile 179
CloseIndex 179
Compileranweisung $F 180
$I 25, 127
$K 25
$R 24
$V 25

DATABASE TOOLBOX 162
Datafile 175
Datei 109
Datei öffnen 117ff, 126f, 177
Datei, index-sequentiell 158ff
sequentiell 110
Datei lesen 120ff, 159, 190, 232
Datei schließen 122, 179, 230
Datei schreiben 118f, 160, 188ff, 232
Dateivariable 114
Datenstruktur 36
DeleteKey 183, 208
DeleteRec 183, 208
Destination 23
Differenzmenge 83f
Directory 27
Dispose 248
Duplicates 175, 219
dynamische Datenstruktur 245
dynamische Variable 246

Editieren 21
Eingabefehler 68f
Elementprüfung 81
EOF 110
EOL 229
Erase 149
EraseFile 213
EraseIndex 213

Feld, eindimensional 36ff
zweidimensional 55ff
Fenstertechnik 233
File 109
FilePos 130
FileSize 133
FillChar 220
FindKey 183, 190
FlushFile 213

FlushIndex 213
Formatfehler 234
Funktionstasten 20

GetRec 183, 191
Grundmenge 79

Heap 246
Hot Keys 20

I/O checking 25
In 81
Include-File 31
Index 40f, 52
Indexdatei 159
Indexfile 175
indizierte Adressierung 41
InitIndex 213
IORESULT 127

Kbd 32, 70
KEYPRESSED 103
Kurzschlußverfahren 32

Laden 21
LongInt 30
Löschalgorithmus 145ff

MakeFile 174f
MakeIndex 175
Matrix 57
MaxDataRecSize 165
MaxDataType 168
MaxHeight 166
MAXINT 102f
MaxKeyLen 165f
MaxKeyType 169
MEMAVAIL 248
Menge 76
Mengenbildung 81ff
Mengenoperationen 80ff
Mengenvereinbarung 77ff

New 247
NextKey 184, 193
NIL 251f
NoDuplicates 175

Odd 83
OpenFile 176
OpenIndex 176
Options 24f
OS shell 27

PageSize 166
PageStackSize 166
physische Datei 116
Pick 28
Platznummer 134
Pointer 245
PrevKey 184, 195
PutRec 183, 207f

Quit 27

Random 102, 104
Randomize 102
Range checking 24
ReadKey 32, 70
Record 91
rename 149f
reset 118ff, 232
rewrite 117, 230

Satzadresse 159f
Satzdatei 158
Satzstatus 169
Satzzeiger 117
Schlüssel 159
Schlüssel, eindeutig 175
Schlüssel, mehrdeutig 175, 218
Schnittmenge 84
SearchKey 184
Seek 134
set 77ff
ShortInt 30
SizeOf 175

Sortieralgorithmus 142ff
Speichern 21
Stack checking 25
statische Datenstruktur 245
String 30

Tabellenverarbeitung 62ff
TABUILD 167
TACCESS.TPU 167f
TAErrorProc 179
TAIOCHECK 179
Textdatei 228ff
Turbo-Access-System 162
typisierte Konstante 46
Typüberprüfung 25

Übersetzen 22
Units 28
Upcase 220
UsedRecs 196
uses 45

Var-string checking 25
Verbund 91ff
Vereinbarung Datei 114ff, 175ff, 229ff
Feld 39, 58
Menge 77ff
Verbund 93
Vereinigungsmenge 83
verkettete Liste 256ff
Anfügen 266ff
Sortieren 287
Verkettung 262ff

Window 234
With 94
Word 30
Write to 26

Zeiger 245ff
Zeigervariable 246
Zugriff Feld 40ff, 59ff
Verbund 93
Zurückschreiben 138

Rudolf Hambusch (Hg.)

GRUNDKURS COMPUTERPRAXIS

Peter Freese
Standard-Betriebssystem MS-DOS
(rororo computer 8145)

Georg Besser/Peter Höver/
Ernst Tiemeyer
Tabellenkalkulation MULTIPLAN
(rororo computer 8146)

Alexander Parkmann
Joachim Röhl
Johannes Verhuven
Programmiersprache BASIC unter MS-DOS
(rororo computer 8147)

Gregor Kuhlmann
Programmiersprache TURBO-PASCAL
(rororo computer 8148)

Peter Freese
Friedrich Müllmerstadt
Datenbanken dBASE
(rororo computer 8140)

Hermann Mehlig
Textverarbeitung WORDSTAR mit MailMerge
(rororo computer 8139)

Joachim Röhl
Johannes Verhuven
Textverarbeitung MS-WORD
(rororo computer 8149)

Gregor Kuhlmann
Programmiersprache TURBO-PASCAL für Fortgeschrittene
(rororo computer 8155)

Johannes Baumann
Stefanie Patzer
Jürgen Schweigert
Integriertes Softwarepaket OPEN ACCESS
(rororo computer 8150)

Alexander Parkmann
Joachim Röhl
Johannes Verhuven
Programmiersprache BASIC für Fortgeschrittene
(rororo computer 8154)

Ernst Tiemeyer
Integriertes Softwarepaket SYMPHONY
(rororo computer 8153)

Heinz Urban/Klaus W. Jamin
Programmiersprache COBOL
(rororo computer 8157)

Bernd Passens/Ingrid Schöll
Textverarbeitung WordPerfect
(rororo computer 8160)

Ute Jacobs/Ursula Riester
Datenbanken Rbase
(rororo computer 8162/
Oktober 1989)

Hermann Mehlig
Integriertes Softwarepaket WORKS
(rororo computer 8163/
November 1989)

Rowohlt

Turbo-Pascal hat sich binnen weniger Jahre zu einer Standardprogrammiersprache entwickelt. Sie bietet, zumal in der Version 4.0, eine Bedieneroberfläche an, die dem Programmierer die Eingabe und Bearbeitung von Programmen sehr erleichtert. Aufbauend auf dem Vorgängerband «Turbo-Pascal» werden in «Turbo-Pascal für Fortgeschrittene» praxisnah, anschaulich und Schritt für Schritt die Datenstrukturen in ihren vielfältigen Formen – ARRAY, SET, RECORD, FILE und POINTER – einschließlich der Fenstertechnik dargelegt.

ro ro ro COMPUTER

DM 16,80